Software Requirements Specification

Project: Supply Locator System

Made by:

Díaz Cervantes Amieva Alejandro

Estrada Zarate Diego Enrique

Morales Anacleto Mario Yair

Torres Iñiguez Ariel

01 - 17th- 2025

TABLE OF CONTENTS

DOCUMENTATION FORM

Date	Revision	Author	Verified Quality dep.
01 - 17 - 2024		Díaz Cervantes Amieva Alejandro Estrada Zarate Diego Enrique Morales Anacleto Mario Yair Torres Iñiguez Ariel	

Document validated by the parties on date:

By the Client	By the Supplying Company
Dr. Ray Brunett Parra Galaviz	
Sgd. Mr./ Mrs.	**Sgd. Mr./ Mrs.**

1. INTRODUCTION

Technology plays a crucial role in improving processes and optimizing resources. While many industries have adopted technological solutions to streamline operations, there are still areas where its potential is not fully realized. One clear example is the construction sector, where the search for and acquisition of materials remains a manual and inefficient process. Despite the presence of numerous suppliers, many consumers still face challenges in quickly and economically finding specific materials. In this context, the development of digital platforms that connect buyers and sellers of construction materials efficiently could significantly transform this process, reducing costs and improving accessibility.

1.1. PURPOSE

The main purpose of this project is to develop a specialized software solution that connects users with local suppliers and buyers of construction materials in an efficient and user-friendly way. This app will allow users to quickly find and purchase materials such as glass, cement, or wood from nearby suppliers. By integrating a real-time map, users can easily locate the closest materials available based on their needs, ensuring faster and more cost-effective purchases. This platform will help reduce transportation costs, enhance convenience, and create a seamless experience for both buyers and sellers.

1.2. SCOPE

The ¨SupplyTrack¨ project will focus on developing an app that allows users to search for construction materials from nearby suppliers and vendors, with an integrated map feature to show the closest locations. The system will allow users to select the material they need, such as glass, wood, or metal, and display nearby shops offering those items. The app will feature real-time updates on material availability, pricing, and delivery options. This project will facilitate efficient material sourcing for construction projects, simplify the buying and selling process, and provide a dynamic, easy-to-use platform for both consumers and suppliers.

1.3. INVOLVED STAFF

Name	Díaz Cervantes Amieva Alejandro
Role	Responsible for gathering the essential data and resources required for the successful development of a correct DataBase.
Professional Category	Student
Responsibilities	DataBase
Contact Information	0323105898@ut-tijuana.edu.mx
Approved	

Name	Estrada Zarate Diego Enrique
Role	Responsible for creating a user interface that is attractive, intuitive and easy to use. It must ensure that the user experience is seamless, allowing them to interact with the buy and sell menu, also with the orders, and map.
Professional Category	Student
Responsibilities	Apps Design
Contact Information	0323105895@ut-tijuana.edu.mx
Approved	

Name	Morales Anacleto Mario Yair
Role	Responsible for designing and managing the organization, storage and processing of data from the map and orders.
Professional Category	Student
Responsibilities	Applied Data Structure
Contact information	0323106025@ut-tijuana.edu.mx
Approved	

Name	Torres Iñiguez Ariel
Role	Responsible for designing, implementing and maintaining the necessary web services and api's for the app.
Professional category	Student
Responsibilities	service-oriented web applications
Contact information	0323105933@ut-tijuana.edu.mx
Approved	

1.4. DEFINITIONS, ACRONYMS AND ABBREVIATIONS

Some words used in this software media may be unfamiliar to most people, even if they are basic, here are some brief descriptions of the words and abbreviations that may be unfamiliar or unknown what they mean completely may be better in major measure to understand this document.

User: A user is a person who uses or interacts with an application, software, electronic device, or computer system to accomplish a specific goal.

Software: This is a term that refers to a program or a set of computer programs, as well as the data, procedures, and guidelines that allow various tasks to be performed in a computer system

Module: A module in the software is a component that performs a specific function and is composed of different computer programs, as well as it can be a class, package, library or even a complete application that intersects with others to form more robust and complete systems.

Database: A database is a software product designed to store large amounts of information in an organized and structured manner. It can be stored locally on your personal computer or on an external remote server.

Reliability: It's the reliability with which an interface can be used by a user, reliability implies that the software must be understandable, learnable, usable, and attractive, and contribute to the functionality and efficiency of the product.

Portability: It's the possibility to compile the source code of a program so that it can be run on different computer platforms. Is the measure of the ease with which an application can transfer a computing environment.

Optimization: Software optimization seeks to adapt computer systems to perform their tasks as efficiently as possible and use as few resources as possible.

Scalability: This refers to the ability of an application or system to handle an increased workload or demand efficiently, without compromising performance.

Security: It is essential to protect privacy and ensure a secure environment for users and for the program itself, protecting the integrity of data and ensuring the safety of users.

Performance: It is the measure of how efficiently the application uses the system resources to do Performance encompasses different aspects of how the program interacts with the underlying device.

1.5. REFERENCES

Reference	Title	Date	Author
https://weremote.net/terminos-basicos-programacion/	21 términos básicos de programación que debes dominar	16 de febrero del 2023	Nicholas Bonder
https://redwerk.es/blog/vocabulario-de-terminos-de-desarrollo-de-software-para-no-tecnicos-los-60-mas-importantes/	Vocabulario de términos de desarrollo de software para no técnicos: Los 60 más importantes	08 de agosto del 2023	REDWERK

1.6. SUMMARY

The goal of this project is to develop a technological platform that will improve the procurement and distribution of construction materials by connecting users with nearby suppliers and buyers. This system is aimed at users who need an efficient, real-time way to locate and purchase construction materials, optimizing both time and cost. By integrating geolocation features and a mobile application, users will be able to search for specific materials, such as glass, cement, or wood, and easily find the closest providers.

The platform will be designed with multiple modules that allow both buyers and sellers to interact seamlessly. These modules will cover material search, location-based mapping, and order management. The solution aims to enhance the overall efficiency of the construction supply chain.

2. GENERAL DESCRIPTION

A technological platform will be developed to streamline the procurement and distribution of construction materials by connecting buyers with nearby suppliers. The system aims to optimize material sourcing, saving both time and cost for users by integrating geolocation features and a mobile application. Users will be able to search for specific materials, such as glass, wood, or cement, and easily locate the nearest vendors. It will also facilitate communication between buyers and sellers, improving the overall efficiency of the construction material supply chain. The program will be organized into different modules that make it simple for users to navigate and understand the system's functionalities.

2.1. PRODUCT PERSPECTIVE

This system targets users who need an efficient and convenient way to source construction materials. The app will be designed to enhance the user experience by offering seamless navigation, search functionality, and instant access to crucial information, all aimed at optimizing material sourcing for construction projects.

2.2. PRODUCT FUNCTIONALITY

The platform will enable users to quickly search for specific materials, view real-time availability, and compare prices from various local suppliers. It will also include features like integrated maps for locating nearby vendors, instant messaging for communication between buyers and sellers, and real-time updates on pricing and stock levels.

PRODUCT FUNCTIONALITY

The system is divided into three types of users: an Administrator with full access, a seller who can register materials, and a buyer who makes the orders.

ADMINISTRATOR:

Create, edit, and delete user accounts.
Approve or reject supplier registrations
Handle disputes between user and suppliers
Moderate reviews and ratings

SELLER USER

Create and manage their supplier profile.
Add, edit, or remove products and services.
Set prices, availability, and service details.
Receive and manage orders from buyers.
Communicate with buyers via messaging.
Track payments and order history.
Respond to customer reviews and ratings.

BUYER USER

Search and browse suppliers and products.
Report issues with orders or suppliers.
Send requests for services or materials.
Receive order confirmations and updates.

2.3. USER CHARACTERISTICS

User Type	Administrator
Education	Background in materials of construction or related fields
Abilities	Basic program management
Activities	Register, configure, and manage all system data, including users information, reports, etc. Having all access.

User Type	Seller
Education	Knowledge in sales, logistics, or construction material management.
Abilities	Inventory management, pricing, and customer service.
Activities	List available construction materials (glass, wood, cement, etc.) with real-time pricing and stock levels.

User Type	Buyer
Education	Knowledge in construction, home improvement, or project management (optional).
Abilities	Searching for products, comparing prices, and managing purchases.
Activities	Search for specific construction materials from nearby suppliers using the app.

2.4. RESTRICTIONS

- The system must have a user friendly and intuitive interface.
- The system should have a data structure based on the standard MySQL model.
- The system must be available 365 days a year.
- The system must be able to handle an increase in the number of sections without compromising its performance or functionality.
- The app should be compatible with the latest version of the client's operating system.

2.5. ASSUMPTIONS AND DEPENDENCE

- The devices on which the software is deployed must have a minimum of resources to function properly.

- Content management and information input should be the responsibility of the company.
- The person responsible for managing the programme should have a basic knowledge of similar software.

2.6. EXPECTED SYSTEM EVOLUTION

The web and technology trends are evolving rapidly. We want to spend time continuing to learn new tools, languages and techniques to help us keep the website up to date and competitive, including adapting our software to cell phones.

3. SPECIFIC REQUIREMENTS

Requirement number	FR 1
Name	User Registration and Authentication
Type	☐ Requirement ☐ Constraint
Description	This module allows users to register and log in via email or social networks. It includes secure authentication and profile management, with a personalized experience with location-based services.
Priority	☐ High/Essential ☐ Medium/wished ☐ Low/Optional

Requirement number	FR 2
Name	Supplier Management
Type	☐ Requirement ☐ Constraint
Description	A module where suppliers can register, manage their profiles, and update information such as services, schedules, pricing, and location. It ensures verified and up-to-date supplier data.
Priority	☐ High/Essential ☐ Medium/wished ☐ Low/Optional

Requirement number	FR 3
Name	Geolocation and Map Integration
Type	☐ Requirement ☐ Constraint
Description	A module that detects the user's location and displays nearby suppliers on a real-time map. It provides route guidance and distance calculation to optimize navigation.
Priority	☐ High/Essential ☐ Medium/wished ☐ Low/Optional

Requirement number	FR 4
Name	Service and Material Requests
Type	☐ Requirement ☐ Constraint
Description	A system where users can request services or materials from suppliers. The module includes request tracking, notifications, and status updates for efficient follow-up.
Priority	☐ High/Essential ☐ Medium/wished ☐ Low/Optionall

Requirement number	FR 5
Name	Buyer-Supplier Communication
Type	☐ Requirement ☐ Constraint
Description	An internal messaging system that allows users and suppliers to communicate directly. This module helps clarify doubts, coordinate deliveries, and discuss service details efficiently.
Priority	☐ High/Essential ☐ Medium/wished ☐ Low/Optional

Requirement number	FR 6
Name	Secure Payment Processing
Type	☐ Requirement ☐ Constraint
Description	A module that integrates PayPal for secure online transactions. It allows users to complete payments directly through the platform and provides transaction history for both users and suppliers.
Priority	☐ High/Essential ☐ Medium/wished ☐ Low/Optional

3.1. COMMON INTERFACE REQUIREMENTS

3.1.1. USER INTERFACES

The interface is a key component of our system, serving as the primary point of interaction between users and the platform. It must be visually appealing, user-friendly, and intuitive to ensure seamless navigation.

The design will prioritize ease of use, allowing users to quickly access essential features such as service and material requests, supplier searches, and internal messaging. Additionally, the interface will be fully responsive, adapting to different screen sizes and devices, including desktops, tablets, and mobile phones.

Our approach focuses on delivering a smooth and efficient user experience, ensuring that users can manage their transactions, locate nearby suppliers, and communicate effectively without complications.

3.1.2. HARDWARE INTERFACES

The system requires interaction with devices that support geolocation services to provide accurate positioning of users and suppliers. It also relies on secure authentication mechanisms to ensure safe and reliable transactions.

3.1.3. SOFTWARE INTERFACES

For the platform to be fully functional, it will integrate with various software components. The system will be accessible through web and mobile applications, compatible with major operating systems such as Windows, macOS, Android, and iOS. Additionally, it will incorporate PayPal for secure online transactions and notifications to keep users informed about request statuses.

3.1.4. COMMUNICATION INTERFACES

The system will facilitate interaction between multiple software components to ensure smooth operations. It will support real-time messaging between users and suppliers, push notifications for request updates, and dynamic maps for supplier visualization. The database will be managed using MySQL, storing critical information such as user profiles, supplier details, service requests, and payment records.

3.2. FUNCTIONAL REQUIREMENTS

User Management and Authentication:

- Registration and login via email and social networks.
- User profile management, including location and preferences.

Geolocation and Map Integration:

- User location detection.
- Display of nearby suppliers on a map.
- Directions and routes to reach suppliers.

Supplier Display and Details:

- Supplier profiles with detailed information: description, services offered, schedules, location, and prices.
- Display of reviews and ratings.

Service or Material Requests:

- System for sending requests or reservations for materials and services.
- Confirmation and tracking of requests through notifications.

User-Supplier Communication:

- Internal messaging system to clarify doubts or coordinate deliveries.
- Administrative Management of Companies or Services:
- Dashboard for managing and validating suppliers.
- Verification of authenticity and content control.
- Secure online payment processing through PayPal.
- Transaction history available for both users and suppliers.

3.3. NON FUNCTIONAL REQUIREMENTS

Security

Total security is guaranteed in the use of the software, safeguarding all stored information and only managed by authorized people without the administration affecting the security of our system.

Reliability

User interfaces will be easy to access, streamlining their understanding, usefulness and functionality.

Availability

The system will be available 24 hours a day, every day of the week.

Maintainability

The developed software will have a user manual that offers all the information necessary for its use and will have monthly updates made by our developers.

Portability

The Production Orders system is compatible with any device with Windows 10 or higher operating system.

4. PROJECT PROGRESS

Login Screen

This React Native code implements a login screen for the SupplyTrack application. The screen handles user authentication using Firebase Authentication. When users enter their email and password, the code validates these credentials against Firebase's authentication system. If the credentials are correct, the user is granted access to the application and their unique user ID (UID) is stored for session management. If the credentials are invalid or there's an error during the authentication process, appropriate error messages are displayed to guide the user.

The screen also provides a user-friendly interface with input fields for email and password, styled with a modern design using orange as the primary color. For users who don't have an account yet, there's a "Sign up" button at the bottom that redirects them to the registration screen.

```
const LoginScreen = ({ navigation }) => {
  const { userloggeduidHandler } = useContext(AuthContext);
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');

  const LoginHandler = async () => {
    if (!email || !password) {
      Alert.alert("Error", "Please enter both email and password.");
      return;
    }

    try {
      const userCredential = await signInWithEmailAndPassword(auth, email, password);
      const user = userCredential.user.uid;
      userloggeduidHandler(user);
      console.log('From LoginScreen (UID)', user);
      Alert.alert("Success", "Logged in successfully!");
    } catch (error) {
      console.error("Error during login:", error.message);
      Alert.alert("Login Failed", error.message);
    }
  };

  return (
    <View style={styles.container}>
      <StatusBar backgroundColor={'#FF3F00'} />
      <View style={{ paddingVertical: 12, width: '95%', alignSelf: 'center', marginBottom: 10 }}>
        <Text style={{ alignSelf: 'center', fontSize: 25, fontWeight: '700' }}>Login</Text>
      </View>

      <View style={styles.inputCont}>
        <FontAwesome name="user" size={24} color="grey" style={styles.icon} />
        <TextInput
          placeholder='Email'
          keyboardType='email-address'
          style={styles.input}
          value={email}
          onChangeText={setEmail}
        />
      </View>
```

```
    <View style={styles.inputCont}>
      <FontAwesome name="lock" size={24} color="grey" style={styles.icon} />
      <TextInput
        placeholder='Password'
        style={styles.input}
        value={password}
        onChangeText={setPassword}
        secureTextEntry={true} // Corregido para ocultar la contraseña
      />
    </View>

    <TouchableOpacity style={styles.loginbutton} onPress={LoginHandler}>
      <Text style={styles.loginbuttonTxt}>Login</Text>
    </TouchableOpacity>

    <View style={{ marginTop: 15, width: '95%', alignSelf: 'center', flexDirection: 'row', justifyContent: 'space-between' }}>
      <View style={{ paddingLeft: 10 }}>
        <Text>Don't have an account?</Text>
      </View>
      <View style={{
        backgroundColor: '#FF3F00',
        borderRadius: 25,
        alignSelf: 'center',
        padding: 10,
        elevation: 2
      }}>
        <TouchableOpacity onPress={() => navigation.navigate('SignUp')}>
          <Text style={{
            fontSize: 17,
            fontWeight: '600',
            color: 'white',
            alignSelf: 'center',
            paddingHorizontal: 10
          }}>Sign up</Text>
        </TouchableOpacity>
      </View>
    </View>
  </View>
);
};
```

Sign Up Screen

The SignupScreen.js code implements the user registration functionality in the SupplyTrack application. When a new user wants to create an account, this screen collects essential information including full name, phone number, email, and password. The code uses Firebase Authentication for user registration and Firestore for storing additional user data.

The registration process follows these steps:
1. First, it validates that all required fields are filled out and ensures the password and confirmation password match.
2. Then, it creates a new user account in Firebase Authentication using the provided email and password.
3. After successful authentication, it creates a new document in the Firestore database under the "Users" collection, storing the user's full name, email, phone number, and account creation timestamp.
4. Once the account is created and data is stored, the user is automatically redirected to the Home screen of the application.

The screen features a user-friendly interface with styled input fields, each accompanied by relevant icons, and maintains the application's consistent design theme using the orange color scheme for the signup button.

```
const SignupScreen = ({ navigation }) => {
    const { userloggeduidHandler } = useContext(AuthContext);
    const [email, setEmail] = useState('');
    const [password, setPassword] = useState('');
    const [cpassword, setCPassword] = useState('');
    const [fullName, setFullName] = useState('');
    const [phone, setPhone] = useState('');

    const createAccountHandler = async () => {
        if (!email || !password || !cpassword || !fullName || !phone) {
            Alert.alert("Error", "Please fill all fields!");
            return;
        }

        if (password !== cpassword) {
            Alert.alert("Error", "Passwords do not match!");
            return;
        }

        try {
            const userCredentials = await createUserWithEmailAndPassword(auth, email, password);
            const user = userCredentials.user;
            const uid = user.uid;

            await new Promise((resolve) => setTimeout(resolve, 1000));

            userloggeduidHandler(uid);

            const userData = {
                fullName: fullName,
                email: email,
                phone: phone,
                createdAt: new Date().toISOString()
            };

            await setDoc(doc(db, "Users", uid), userData);

            console.log("✅ Account Created & Data Saved. UID:", uid);
            Alert.alert("Success", "Account created successfully!");

            navigation.replace("Home");

        } catch (error) {
            console.error("❌ Error creating account:", error.message);
```

```
    return (
        <View style={styles.container}>
            <StatusBar backgroundColor={'#FF3F00'} />
            <View style={{ paddingVertical: 12, width: '95%', alignSelf: 'center', marginBottom: 10 }}>
                <Text style={{ alignSelf: 'center', fontSize: 25, fontWeight: '700' }}>Sign up</Text>
            </View>

            <View style={styles.inputCont}>
                <FontAwesome name="user" size={24} color="grey" style={styles.icon} />
                <TextInput
                    placeholder='Full Name'
                    style={styles.input}
                    value={fullName}
                    onChangeText={setFullName}
                />
            </View>

            <View style={styles.inputCont}>
                <FontAwesome name="phone" size={24} color="grey" style={styles.icon} />
                <TextInput
                    placeholder='Phone'
                    keyboardType='phone-pad'
                    style={styles.input}
                    value={phone}
                    onChangeText={setPhone}
                />
            </View>

            <View style={styles.inputCont}>
                <FontAwesome name="envelope" size={24} color="grey" style={styles.icon} />
                <TextInput
                    placeholder='Email'
                    keyboardType='email-address'
                    style={styles.input}
                    value={email}
                    onChangeText={setEmail}
                />
            </View>
```

```
            <View style={styles.inputCont}>
                <FontAwesome name="envelope" size={24} color="grey" style={styles.icon} />
                <TextInput
                    placeholder='Email'
                    keyboardType='email-address'
                    style={styles.input}
                    value={email}
                    onChangeText={setEmail}
                />
            </View>

            <View style={styles.inputCont}>
                <FontAwesome name="lock" size={24} color="grey" style={styles.icon} />
                <TextInput
                    placeholder='Password'
                    style={styles.input}
                    secureTextEntry
                    value={password}
                    onChangeText={setPassword}
                />
            </View>

            <View style={styles.inputCont}>
                <FontAwesome name="lock" size={24} color="grey" style={styles.icon} />
                <TextInput
                    placeholder='Confirm Password'
                    style={styles.input}
                    secureTextEntry
                    value={cpassword}
                    onChangeText={setCPassword}
                />
            </View>

            <TouchableOpacity style={styles.loginbutton} onPress={createAccountHandler}>
                <Text style={styles.loginbuttonTxt}>Sign up</Text>
            </TouchableOpacity>
        </View>
    );
};
```

Sign Up Next Screen

The SignupNextScreen.js code represents an additional profile completion step in the SupplyTrack application's registration process. This screen appears after the initial signup process to collect additional user information.

The screen provides a simple form interface where users can input:
1. Their name
2. Phone number (with a specialized keyboard for phone number input)
3. Address information
After filling out these additional details, when the user clicks the "Next" button, the application:

1. Shows a success message confirming the account creation
2. Navigates the user to the HomeScreen of the application
The screen maintains the application's consistent design language, using the same orange color scheme for the status bar and button, and similar styling for input fields with rounded corners. However, it's worth noting that there's some commented-out code that suggests a planned "Already have an account?" section that would redirect to the login screen, but this functionality is currently disabled.

```
const SignupNextScreen = ({ navigation }) => {
  return (
    <View style={styles.container}>
      <StatusBar backgroundColor={'#FF3F00'} />
      <View style={{ paddingVertical: 12, width: '95%', alignSelf: 'center', marginBottom: 10 }}>
        <Text style={{ alignSelf: 'center', fontSize: 25, fontWeight: '700' }}>Complete Profile</Text>
      </View>

      <TextInput
        placeholder='Name'
        style={styles.input}
      />
      <TextInput
        placeholder='Phone'
        keyboardType='phone-pad'
        style={styles.input}
      />
      <TextInput
        placeholder='Address'
        style={styles.input}
      />

      <TouchableOpacity style={styles.loginbutton} onPress={() => {
        Alert.alert("Success", "Account created successfully!");
        navigation.navigate("HomeScreen");
      }}>
        <Text style={styles.loginbuttonTxt}>Next</Text>
      </TouchableOpacity>
    </View>
  );
};
```

Track Order Items

1- The component imports the necessary dependencies from React Native (such as FlatList, StyleSheet, Text, View, Image) and Firebase to interact with the Firestore database. AuthContext is also imported to access the UID of the authenticated user.

2- UseContext(AuthContext) is used to obtain the UID of the authenticated user, which will be used to query Firestore and obtain user and order specific data.

3- Two states are defined with useState: orderData to store the order items and user to store the user's data, such as his profile or balance. Both states are updated with the Firebase data.

4- The getuserData function queries the Firestore UserData collection using the user's UID. If it finds user data, it updates the user state. If not, it displays a message in the console.

5- The fetchData function queries the order items from the OrderItems collection, using the order ID. It uses onSnapshot to listen for real-time changes and update orderData with the order items.

6- The getDta function takes a product ID and filters the foodDataAll array to get the corresponding product details. This ensures that the product data associated with the order is displayed correctly.

7- The component traverses orderData and uses FlatList to render the list of items. Each item displays a card with the image, name, price and quantity requested, obtaining the complete data using getDta.

8- By using onSnapshot to listen for changes in real time, the component ensures that the interface is automatically updated when order or user data changes, providing an interactive and dynamic experience.

```
import { FlatList, StyleSheet, Text, View, Image } from 'react-native'
import React, { useContext, useEffect, useState } from 'react'
import { firebase } from '../Firebase/FirebaseConfig'
import { AuthContext } from '../Context/AuthContext';


const TrackOrderItems = ({ foodDataAll, data, navigation }) => {
    const { userloggeduid } = useContext(AuthContext);
    const [orderData, setOrderData] = useState([]);


    const [user, setUser] = useState([]);

    const getuserData = async () => {
        const docRef = firebase.firestore().collection('UserData').where('uid', '==', userloggeduid)
        const doc = await docRef.get();
        if (!doc.empty) {
            doc.forEach((doc) => {
                setUser(doc.data());
            })
        }
        else {
            console.log('no user data');
        }
    }

    useEffect(() => {

        getuserData();
    }, [userloggeduid]);

    // console.log('user is ', user.totalCoin - 5)

    useEffect(() => {
        // Fetch data from Firebase
        const fetchData = async () => {
            const foodRef = firebase.firestore().collection('OrderItems').doc(data);

            foodRef.onSnapshot(doc => {
                // setOrderData(snapshot.docs.map(doc => doc.data().cartItems))
```

```
                // console.log('dekh veere',doc.data() )
                setOrderData(doc.data().cartItems)
            }
            )
        };

        fetchData();
    }, [data]);

    const getDta = (id) => {
        const nData = foodDataAll.filter((items) => items.id === id)
        return nData;
    }



    // console.log('ye dekh bhai222', foodDataAll)
    return (
        <View>
            {/* <Text>Ye dekh veer</Text> */}

            {orderData && orderData.map((order, index) => (
                <View key={index}
                //  style={{ borderRadius: 20, backgroundColor: '#f2f2f2', width: '95%', alignSelf: 'center', marginVe
                >


                <FlatList

                    data={getDta(order.item_id)}

                    renderItem={
                        ({ item }) => {
                            console.log('ye dekh veer23123123', item)
```

```
                renderItem={
                    ({ item }) => {
                        console.log('ye dekh veer23123123', item)
                        return (
                            <View style={styles.orderItemContainer}>
                                <View>
                                    <Image source={{ uri: item.FoodImageUrl }} style={styles.cardimage} />
                                </View>

                                <View style={styles.orderItemContainer_2}>
                                    <View>
                                        <Text style={styles.orderItemName}>{item.FoodName}</Text>
                                        <Text style={styles.orderItemPrice} >${item.FoodPrice}</Text>
                                        <Text>Qty : {order.FoodQuantity} unit </Text>

                                    </View>
                                </View>
                            </View>
                        )
                    }
                }


            />
        </View>
        ))}
    </View>
  )
}
```

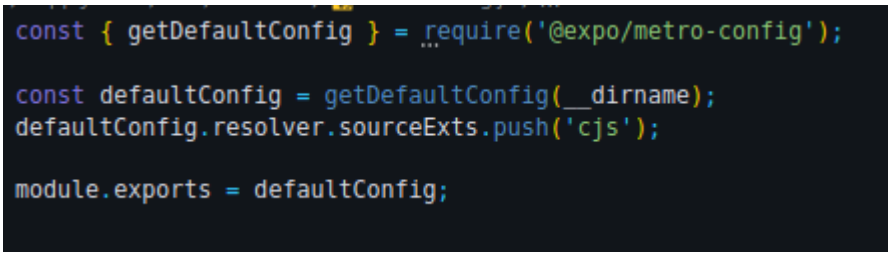
```
export default TrackOrderItems

const styles = StyleSheet.create({
    orderItemContainer: {
        flexDirection: 'row',
        backgroundColor: 'green',
        marginVertical: 2,
        width: '95%',
        alignSelf: 'center',
        borderRadius: 20,
        backgroundColor: '#f2f2f2',
        elevation: 2
    },
    cardimage: {
        width: 90,
        height: 80,
        borderBottomLeftRadius: 20,
        borderTopLeftRadius: 20
    },
    orderItemContainer_2: {
        paddingHorizontal: 10,
        flexDirection: 'row',
        justifyContent: 'space-between',

    },
    orderItemName: {
        fontSize: 16,
        fontWeight: '600'
    },
})
```

Metro Config

This code imports the getDefaultConfig function from the @expo/metro-config package.
It then creates a constant called defaultConfig that calls the getDefaultConfig(__dirname) function and passes it the __dirname parameter, which is a variable in Node.js that represents the path to the current directory. This function returns a configuration object that is tailored for the project.

Then, defaultConfig.resolve.sourceExts.push('cjs'); modifies the resolve.sourceExts property of the configuration. sourceExts is an array that defines the file extensions that Metro should recognize. By adding 'cjs' to the array, Metro is instructed to also include files with the .cjs extension (CommonJS modules) in the packaging process.

Finally, export the modified configuration with module.exports = defaultConfig.



```
const { getDefaultConfig } = require('@expo/metro-config');

const defaultConfig = getDefaultConfig(__dirname);
defaultConfig.resolver.sourceExts.push('cjs');

module.exports = defaultConfig;
```

Firebase Configur

This code first makes the imports:

- initializeApp: Imported to initialize the Firebase application.
- getAuth and initializeAuth: Used to handle user authentication.
- getReactNativePersistence: This function allows to configure the authentication persistence in React Native.
- getFirestore: It is very important, since it allows interacting with the Firebase database.
- ReactNativeAsyncStorage: It is a library that provides asynchronous storage in React Native, similar to localStorage on the web.

Then, an object (firebaseConfig) is defined that contains the configuration needed to connect the application to Firebase. For example, apiKey is one of the credentials that Firebase provides us with when we create a project in the Firebase console.

Then, Firebase is initialized by passing the firebaseConfig object to the initializeApp function. This creates a Firebase instance (app) that will be used to access Firebase services.

Next, the Firebase authentication service is initialized with initializeAuth. Here you use getReactNativePersistence(ReactNativeAsyncStorage) to configure authentication persistence in React Native using AsyncStorage. This allows the authentication state (e.g., whether a user is logged in) to persist even after the app is closed. For example, if a user logs in and closes the app, when they reopen the app they will still be authenticated.

```
import { initializeApp } from "firebase/app";
import { getAuth, initializeAuth, getReactNativePersistence } from "firebase/auth";
import { getFirestore } from "firebase/firestore";
import ReactNativeAsyncStorage from '@react-native-async-storage/async-storage';

const firebaseConfig = {
  apiKey: "AIzaSyD5TGmTdHh3SPX95mbGYaXr_VOsi5_prSw",
  authDomain: "supplytrack-d70f3.firebaseapp.com",
  projectId: "supplytrack-d70f3",
  storageBucket: "supplytrack-d70f3.appspot.com",
  messagingSenderId: "944128623627",
  appId: "1:944128623627:web:99357b3bfb5ecefae9730e",
  measurementId: "G-1S50JG2N9G"
};

// Inicializar Firebase
const app = initializeApp(firebaseConfig);

// Inicializar Auth
const auth = initializeAuth(app, {
  persistence: getReactNativePersistence(ReactNativeAsyncStorage)
});

// Inicializar Firestore
const db = getFirestore(app);

export { auth, db };
```

Auth Context

This code first imports some React components, such as useEffect, useState and createContext, which are tools to manage the state and lifecycle of components. The AsyncStorage library is also imported, which allows to store data persistently in React Native, such as the user's UID. In addition, the Firebase authentication instance (auth) that we configured earlier is imported, along with Firebase's onAuthStateChanged and signOut functions. onAuthStateChanged is used to listen for changes in the authentication state, such as when a user logs in or logs out, and signOut allows the user to be logged out.

Next, a context is created using createContext. This context, called AuthContext, allows you to share authentication state (such as the user's UID) and related functions (e.g., logout) throughout the application.

Next, the AuthProvider component, which is a context provider, is defined. This component wraps other components (that is why it receives children) and provides the authentication context to the whole application. Within AuthProvider, two states are defined:

- **userloggeduid**: It is a state that stores the UID of the authenticated user.
- **loading**: It is a state that indicates if the application is loading, for example, verifying if the user is already authenticated.

```jsx
import React, { useEffect, useState, createContext } from "react";
import AsyncStorage from "@react-native-async-storage/async-storage";
import { auth } from "../Firebase/FirebaseConfig";
import { onAuthStateChanged, signOut } from "firebase/auth";

const AuthContext = createContext();

const AuthProvider = ({ children }) => {
    const [userloggeduid, setUserloggeduid] = useState(null);
    const [loading, setLoading] = useState(true);

    // Función para establecer el UID en el estado y AsyncStorage
    const userloggeduidHandler = async (userid) => {
        try {
            setUserloggeduid(userid);
            await AsyncStorage.setItem("userloggeduid", userid);
        } catch (error) {
            console.error("Error saving user UID to AsyncStorage:", error);
        }
    };

    // Verificar si el usuario está autenticado al iniciar la app
    const checkIsLogged = async () => {
        try {
            const value = await AsyncStorage.getItem("userloggeduid");
            if (value) {
                console.log("User logged UID retrieved from AsyncStorage:", value);
                setUserloggeduid(value);
            } else {
                console.log("User logged UID not found in AsyncStorage");
            }
        } catch (error) {
            console.error("Error retrieving userloggeduid:", error);
        }
    };

    // Escuchar cambios en la autenticación de Firebase
    useEffect(() => {
        const unsubscribe = onAuthStateChanged(auth, async (user) => {
            if (user) {
```

```jsx
const AuthProvider = ({ children }) => {
    // Escuchar cambios en la autenticación de Firebase
    useEffect(() => {
        const unsubscribe = onAuthStateChanged(auth, async (user) => {
            if (user) {
                console.log("Firebase Auth detected user:", user.uid);
                userloggeduidHandler(user.uid);
            } else {
                console.log("No user detected, clearing session.");
                setUserloggeduid(null);
                await AsyncStorage.removeItem("userloggeduid");
            }
            setLoading(false);
        });

        return () => unsubscribe();
    }, []);

    // Función para cerrar sesión
    const logout = async () => {
        try {
            await signOut(auth);
            await AsyncStorage.removeItem("userloggeduid");
            setUserloggeduid(null);
            console.log("Sesión cerrada exitosamente.");
        } catch (error) {
            console.error("Error al cerrar sesión:", error.message);
        }
    };

    console.log("From Context (UID)", userloggeduid);

    return (
        <AuthContext.Provider value={{ userloggeduid, userloggeduidHandler, checkIsLogged, logout }}>
            {!loading && children} {/* No renderiza `children` hasta que `loading` sea falso */}
        </AuthContext.Provider>
    );
};

export { AuthProvider, AuthContext };
```

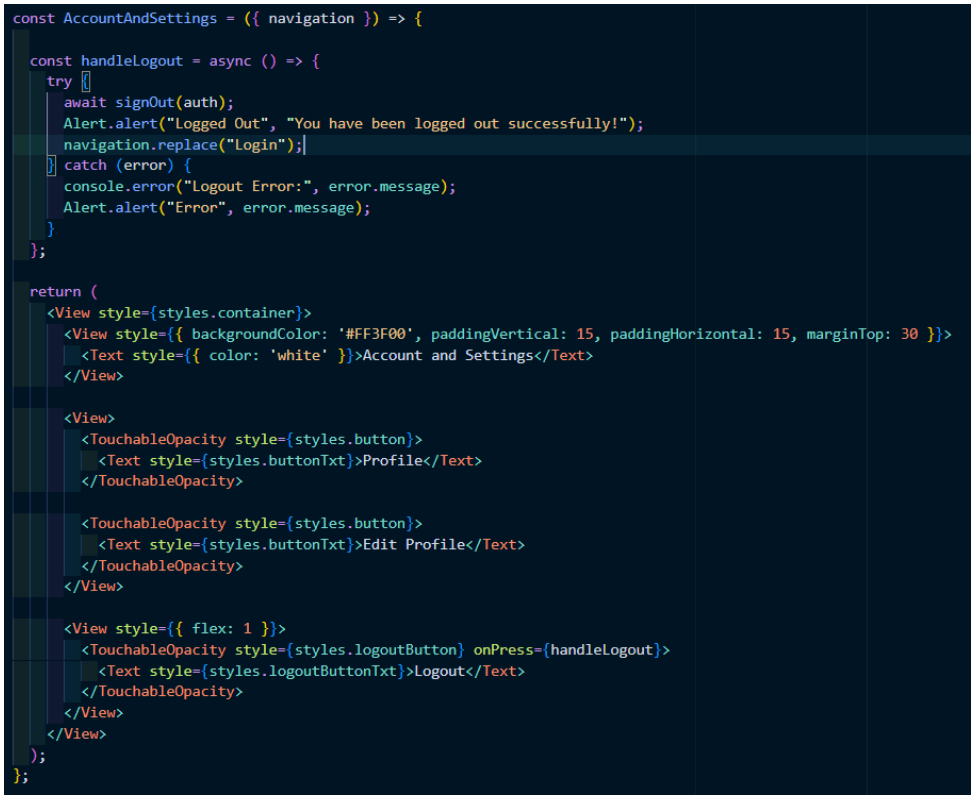
Account and Settings

AccountAndSettings.js code implements the account management screen in the SupplyTrack application. This screen provides users with access to their account settings and profile management options.

The screen contains three main functionalities:

1. A header section displaying "Account and Settings" with the application's signature orange color
2. Two main navigation buttons:
    - Profile: (Currently a placeholder button for viewing profile details)
    - Edit Profile: (Currently a placeholder button for editing profile information)
3. A logout function that:
    - Signs out the user from Firebase Authentication
    - Displays a success message when logout is completed
    - Redirects the user back to the Login screen
    - Handles any potential errors during the logout process with appropriate error messages

The screen maintains the application's consistent design language with styled buttons and a strategically placed logout button at the bottom of the screen. The logout button is intentionally styled differently to distinguish it from the main action buttons.



```
const AccountAndSettings = ({ navigation }) => {

  const handleLogout = async () => {
    try {
      await signOut(auth);
      Alert.alert("Logged Out", "You have been logged out successfully!");
      navigation.replace("Login");
    } catch (error) {
      console.error("Logout Error:", error.message);
      Alert.alert("Error", error.message);
    }
  };

  return (
    <View style={styles.container}>
      <View style={{ backgroundColor: '#FF3F00', paddingVertical: 15, paddingHorizontal: 15, marginTop: 30 }}>
        <Text style={{ color: 'white' }}>Account and Settings</Text>
      </View>

      <View>
        <TouchableOpacity style={styles.button}>
          <Text style={styles.buttonTxt}>Profile</Text>
        </TouchableOpacity>

        <TouchableOpacity style={styles.button}>
          <Text style={styles.buttonTxt}>Edit Profile</Text>
        </TouchableOpacity>
      </View>

      <View style={{ flex: 1 }}>
        <TouchableOpacity style={styles.logoutButton} onPress={handleLogout}>
          <Text style={styles.logoutButtonTxt}>Logout</Text>
        </TouchableOpacity>
      </View>
    </View>
  );
};
```

Home Screen

The HomeScreen.js code implements the main landing page of the SupplyTrack application. This screen serves as the central hub for users to browse and search for construction materials.

The screen includes several key features:

1. Location Services:
    - Requests user permission to access location
    - Displays the current city and country
    - Automatically fetches and updates location information
2. Material Management:
    - Fetches material data from Firebase Firestore
    - Implements a search functionality that filters materials by name or supplier
    - Displays materials in an organized card slider format
3. User Interface Components:
    - A header bar showing the current location
    - A search box with an icon for easy material searching
    - Categories section for material classification
    - An offer slider for promotional content
    - A card slider displaying all available materials
4. Authentication Check:
    - Verifies user authentication status
    - Ensures only authenticated users can access the material data
    - Logs authentication status for debugging purposes

```
const HomeScreen = ({ navigation }) => {
  const [materialData, setMaterialData] = useState([]);
  const [filteredData, setFilteredData] = useState([]);
  const [searchQuery, setSearchQuery] = useState('');
  const [locationName, setLocationName] = useState("Fetching location...");

  const scrollViewRef = useRef(null);
  const categoryRefs = useRef({});

  useEffect(() => {
    checkUserAuthentication();
    requestLocationPermission();
  }, []);

  const checkUserAuthentication = async () => {
    const auth = getAuth();
    if (!auth.currentUser) {
      console.warn("⚠️ No authenticated user found!");
      return;
    }
    console.log("✅ Authenticated User UID:", auth.currentUser.uid);
    fetchMaterialData();
  };

  const fetchMaterialData = async () => {
    try {
      console.log("🔥 Fetching materials from Firestore...");
      const querySnapshot = await getDocs(collection(db, "Materials"));

      if (querySnapshot.empty) {
        console.warn("⚠️ No materials found in Firestore.");
        return;
      }

      const data = querySnapshot.docs.map(doc => ({
        id: doc.id,
        ...doc.data()
      }));

      setMaterialData(data);
      setFilteredData(data);
      console.log("✅ Materials Loaded:", data);

    } catch (error) {
      console.error("❌ Error fetching material data:", error);
```

```
const requestLocationPermission = async () => {
  const { status } = await Location.requestForegroundPermissionsAsync();
  if (status !== 'granted') {
    console.log('❌ Permission to access location was denied');
    return;
  }
  getLocation();
};

const getLocation = async () => {
  try {
    const location = await Location.getCurrentPositionAsync({});
    const { latitude, longitude } = location.coords;
    console.log('📍 Latitude:', latitude);
    console.log('📍 Longitude:', longitude);
    const locName = await getLocationName(latitude, longitude);
    setLocationName(locName || "Unknown location");
  } catch (error) {
    console.log('❌ Error getting location:', error);
  }
};

const getLocationName = async (latitude, longitude) => {
  try {
    const geocode = await Location.reverseGeocodeAsync({ latitude, longitude });
    if (geocode.length > 0) {
      return `${geocode[0].city}, ${geocode[0].country}`;
    }
  } catch (error) {
    console.log('❌ Error fetching location name:', error);
  }
  return null;
};
```

```
const handleSearch = (text) => {
  setSearchQuery(text);
  if (text === '') {
    setFilteredData(materialData);
  } else {
    const filtered = materialData.filter(item =>
      item.MaterialName.toLowerCase().includes(text.toLowerCase()) ||
      item.Supplier.toLowerCase().includes(text.toLowerCase())
    );
    setFilteredData(filtered);
  }
};

const scrollToCategory = (category) => {
  if (categoryRefs.current[category]) {
    categoryRefs.current[category].measureLayout(scrollViewRef.current, (x, y) => {
      scrollViewRef.current.scrollTo({ y, animated: true });
    });
  }
};
```

```
return (
  <View style={styles.mainContainer}>
    <StatusBar backgroundColor={'#FF3F00'} />
    <Headerbar location={locationName} />

    <View style={styles.searchbox}>
      <AntDesign name="search1" size={24} color="black" />
      <TextInput
        style={styles.input}
        placeholder="Search materials"
        value={searchQuery}
        onChangeText={handleSearch}
      />
    </View>

    <ScrollView ref={scrollViewRef} style={styles.scrollContainer} showsVerticalScrollIndicator={false}>
      <Categories onCategoryPress={scrollToCategory} />
      <OfferSlider />
      <CardSlider navigation={navigation} data={filteredData} categoryRefs={categoryRefs} />
    </ScrollView>
  </View>
);
};
```

Product Screen

The ProductScreen.js code implements the detailed product view in the SupplyTrack application. This screen is displayed when a user selects a specific construction material to view or purchase.

The screen provides several key functionalities:

1. Product Display:
    - Shows a large image of the selected material
    - Displays essential product information including:
        - Material name
        - Price in MXN
        - Category
        - Supplier details
2. Quantity Management:
    - Interactive quantity selector with plus and minus buttons
    - Direct input field for quantity
    - Prevents selecting quantities less than 1
3. Cart Integration:
    - Adds selected materials to the user's cart
    - Handles both new items and existing items:
        - Updates quantity if item already exists in cart
        - Adds new entry if item is not in cart
    - Stores cart data in Firebase Firestore
    - Provides feedback through success/error alerts
4. Navigation:
    - Includes a close button to return to the home screen
    - Automatically redirects to home screen if no product data is available

-

```
const ProductScreen = ({ navigation, route }) => {
    const { userloggeduid } = useContext(AuthContext);
    const [quantity, setQuantity] = useState('1');
    const data = route.params;

    if (!data) {
        navigation.navigate('HomeScreen');
        return null;
    }

    const AddToCartHandler = async () => {
        const date = new Date().getTime().toString();
        const docRef = doc(db, 'UserCart', userloggeduid);
        const materialData = {
            item_id: data.id,
            MaterialQuantity: parseInt(quantity, 10),
            userid: userloggeduid,
            cartItemId: date + userloggeduid,
            totalMaterialPrice: parseInt(data.MaterialPrice) * parseInt(quantity),
            MaterialName: data.MaterialName,
            MaterialImageUrl: data.MaterialImageUrl,
            Supplier: data.Supplier
        };

        try {
            const docSnap = await getDoc(docRef);
            if (docSnap.exists()) {
                const cartItems = docSnap.data().cartItems || [];
                const existingItemIndex = cartItems.findIndex((item) => item.item_id === data.id);

                if (existingItemIndex !== -1) {
                    cartItems[existingItemIndex].MaterialQuantity += parseInt(quantity, 10);
                    await updateDoc(docRef, { cartItems });
                } else {
                    await updateDoc(docRef, { cartItems: arrayUnion(materialData) });
                }
            } else {
                await setDoc(docRef, { cartItems: [materialData] });
            }

            Alert.alert("Success", "Material added to cart!");
        } catch (error) {
            console.error('Error adding to cart:', error);
            Alert.alert("Error", error.message);
        }
    };
```

```
return (
    <ScrollView style={styles.container}>
        <StatusBar backgroundColor={'#FF3F00'} />

        {/* Header */}
        <View style={styles.header}>
            <TouchableOpacity onPress={() => navigation.navigate('HomeScreen')}>
                <Text style={styles.headerText}>Close</Text>
            </TouchableOpacity>
        </View>

        {/* Contenido */}
        <View style={styles.containerIn}>
            <Image source={{ uri: data.MaterialImageUrl }} style={styles.cardimage} />

            <Text style={styles.materialName}>{data.MaterialName}</Text>
            <Text style={styles.materialPrice}>${data.MaterialPrice} MXN</Text>

            <Text style={styles.categoryText}>Category: {data.MaterialType}</Text>
            <Text style={styles.categoryText}>Supplier: {data.Supplier}</Text>

            <Text style={styles.quantityLabel}>Quantity</Text>
            <View style={styles.quantityContainer}>
                <TouchableOpacity
                    style={styles.quantityButton}
                    onPress={() => setQuantity((Math.max(parseInt(quantity) - 1, 1)).toString())}
                >
                    <Text style={styles.quantityButtonText}>-</Text>
                </TouchableOpacity>

                <TextInput
                    style={styles.quantityInput}
                    value={quantity}
                    keyboardType="numeric"
                    onChangeText={setQuantity}
                />

                <TouchableOpacity
                    style={styles.quantityButton}
                    onPress={() => setQuantity((parseInt(quantity) + 1).toString())}
                >
                    <Text style={styles.quantityButtonText}>+</Text>
                </TouchableOpacity>
```

User Cart Screen

The UserCartScreen.js serves as the shopping cart interface in the SupplyTrack application, providing users with a comprehensive view of their selected construction materials. When users access this screen, it automatically connects to Firebase Firestore to retrieve and display their current cart items. Each item is presented in a visually appealing card format showing the material's image, name, price in Mexican Pesos, and the selected quantity.

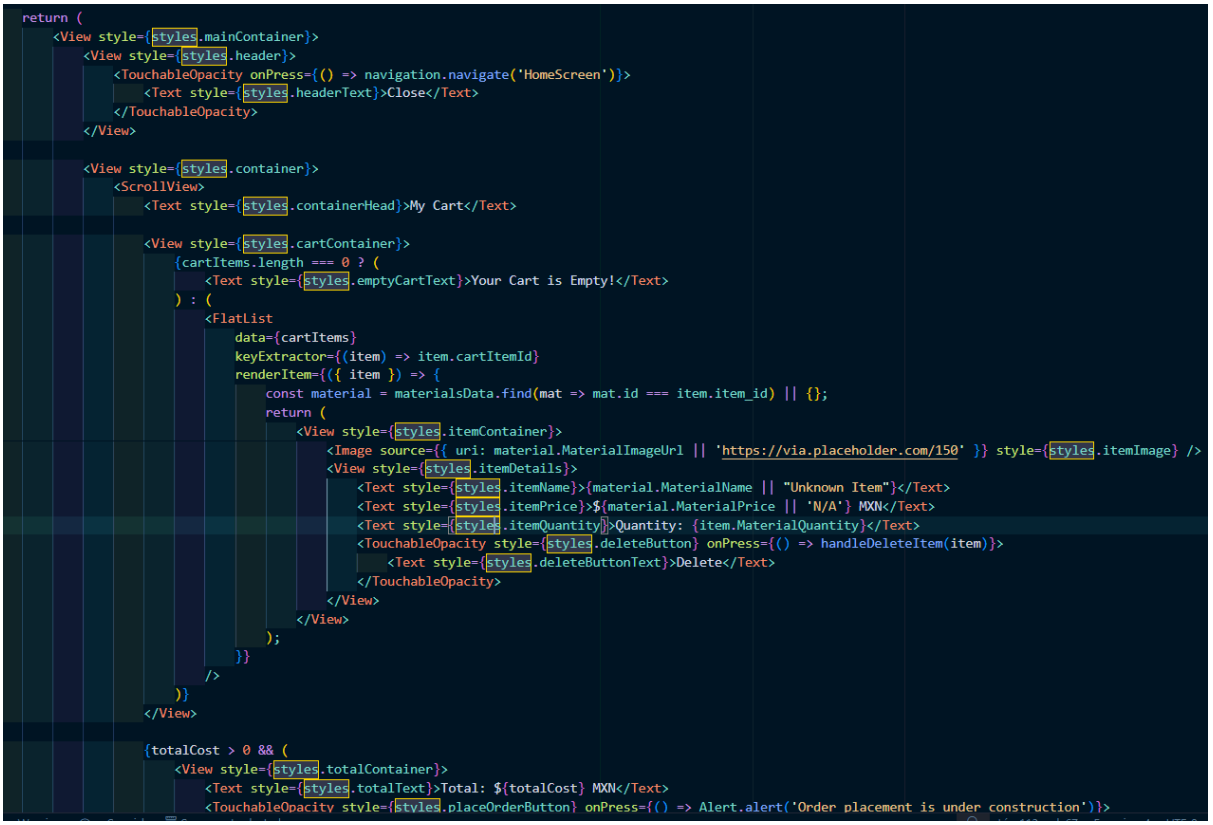
The screen actively maintains data accuracy by synchronizing with the database whenever it comes into focus, ensuring users always see the most current information. Users can easily manage their cart by removing individual items, with the interface updating immediately to reflect these changes. The system automatically recalculates the total cost whenever items are modified or removed.

For user convenience, the interface includes a scrollable list of all cart items, with a clear indication when the cart is empty. At the bottom of the screen, users can see their total purchase amount, and although currently under construction, there's a "Place Order" button for future implementation of the checkout process.

The screen maintains the application's design language with its signature orange theme and provides intuitive navigation with a close button that returns users to the home screen. All user actions receive immediate visual feedback, creating a smooth and responsive shopping experience.

```
const UserCartScreen = ({ navigation }) => {
    const { userloggeduid } = useContext(AuthContext);
    const [cartItems, setCartItems] = useState([]);
    const [materialsData, setMaterialsData] = useState([]);
    const [totalCost, setTotalCost] = useState(0);

    const fetchCartData = async () => {
        try {
            const cartRef = doc(db, 'UserCart', userloggeduid);
            const cartSnap = await getDoc(cartRef);
            if (cartSnap.exists()) {
                setCartItems(cartSnap.data().cartItems || []);
            } else {
                setCartItems([]);
                console.log('Cart is empty.');
            }
        } catch (error) {
            console.error('Error fetching cart data:', error);
        }
    };

    const fetchMaterialsData = async () => {
        try {
            const materialsRef = collection(db, "Materials");
            const querySnapshot = await getDocs(materialsRef);
            const materialsArray = querySnapshot.docs.map(doc => ({
                id: doc.id,
                ...doc.data()
            }));
            setMaterialsData(materialsArray);
        } catch (error) {
            console.error("Error fetching materials data:", error);
        }
    };

    const calculateTotalCost = () => {
        let total = cartItems.reduce((sum, item) => sum + (item.totalMaterialPrice || 0), 0);
        setTotalCost(total);
    };
```

```
    const handleDeleteItem = async (item) => {
        try {
            const cartRef = doc(db, 'UserCart', userloggeduid);
            await updateDoc(cartRef, {
                cartItems: arrayRemove(item)
            });
            setCartItems(prevItems => prevItems.filter(cartItem => cartItem.cartItemId !== item.cartItemId));
        } catch (error) {
            console.error('Error deleting item:', error);
        }
    };

    const clearCart = async () => {
        try {
            const cartRef = doc(db, 'UserCart', userloggeduid);
            await deleteDoc(cartRef);
            setCartItems([]);
        } catch (error) {
            console.error("Error deleting cart:", error);
        }
    };

    useEffect(() => {
        fetchCartData();
        fetchMaterialsData();
    }, []);

    useEffect(() => {
        calculateTotalCost();
    }, [cartItems]);

    useFocusEffect(
        React.useCallback(() => {
            fetchCartData();
            calculateTotalCost();
        }, [])
    );
```



```
return (
    <View style={styles.mainContainer}>
        <View style={styles.header}>
            <TouchableOpacity onPress={() => navigation.navigate('HomeScreen')}>
                <Text style={styles.headerText}>Close</Text>
            </TouchableOpacity>
        </View>

        <View style={styles.container}>
            <ScrollView>
                <Text style={styles.containerHead}>My Cart</Text>

                <View style={styles.cartContainer}>
                    {cartItems.length === 0 ? (
                        <Text style={styles.emptyCartText}>Your Cart is Empty!</Text>
                    ) : (
                        <FlatList
                            data={cartItems}
                            keyExtractor={(item) => item.cartItemId}
                            renderItem={({ item }) => {
                                const material = materialsData.find(mat => mat.id === item.item_id) || {};
                                return (
                                    <View style={styles.itemContainer}>
                                        <Image source={{ uri: material.MaterialImageUrl || 'https://via.placeholder.com/150' }} style={styles.itemImage} />
                                        <View style={styles.itemDetails}>
                                            <Text style={styles.itemName}>{material.MaterialName || "Unknown Item"}</Text>
                                            <Text style={styles.itemPrice}>${material.MaterialPrice || 'N/A'} MXN</Text>
                                            <Text style={styles.itemQuantity}>Quantity: {item.MaterialQuantity}</Text>
                                            <TouchableOpacity style={styles.deleteButton} onPress={() => handleDeleteItem(item)}>
                                                <Text style={styles.deleteButtonText}>Delete</Text>
                                            </TouchableOpacity>
                                        </View>
                                    </View>
                                );
                            }}
                        />
                    )}
                </View>

                {totalCost > 0 && (
                    <View style={styles.totalContainer}>
                        <Text style={styles.totalText}>Total: ${totalCost} MXN</Text>
                        <TouchableOpacity style={styles.placeOrderButton} onPress={() => Alert.alert('Order placement is under construction')}>
```

User Profile

The UserProfile.js implements a personal profile management interface in the SupplyTrack application, allowing users to view and update their personal information. When users access their profile, the screen immediately retrieves their current information from Firebase Firestore, displaying their full name, email address, and phone number in a clean, organized layout.

The interface incorporates a smart editing system that initially displays user information in read-only mode, with an "Edit Profile" button that enables users to modify their details. During editing, users can update their full name and phone number, while the email address remains non-editable for security purposes. The system provides real-time feedback through loading indicators and success/error messages when changes are made.

For user convenience, the screen features a straightforward layout with clearly labeled input fields, each accompanied by relevant icons. The interface maintains the application's design consistency with the signature orange theme and rounded input fields. When users make changes to their profile, they can save their updates with a single tap, and the system immediately syncs these changes with the Firebase database.

```
const UserProfile = () => {
    const { userloggeduid } = useContext(AuthContext);
    const [userData, setUserData] = useState(null);
    const [loading, setLoading] = useState(true);
    const [isEditing, setIsEditing] = useState(false);

    const [editedData, setEditedData] = useState({
        fullName: '',
        email: '',
        phone: ''
    });

    const fetchUserData = async () => {
        if (!userloggeduid) return;

        try {
            const userDoc = await getDoc(doc(db, "Users", userloggeduid));
            if (userDoc.exists()) {
                setUserData(userDoc.data());
                setEditedData(userDoc.data());
            } else {
                console.log('No user profile found.');
            }
        } catch (error) {
            console.error("Error fetching user data:", error);
        } finally {
            setLoading(false);
        }
    };

    useEffect(() => {
        fetchUserData();
    }, []);

    const handleSave = async () => {
        if (!userloggeduid) return;

        try {
            const userRef = doc(db, "Users", userloggeduid);
            await updateDoc(userRef, {
                fullName: editedData.fullName,
                phone: editedData.phone
            });
```

```
            setUserData(editedData);
            setIsEditing(false);
            Alert.alert("Success", "Profile updated successfully!");
        } catch (error) {
            console.error("Error updating profile:", error);
            Alert.alert("Error", "Could not update profile.");
        }
    };

    if (loading) {
        return (
            <View style={styles.loadingContainer}>
                <ActivityIndicator size="large" color="#FF3F00" />
                <Text>Loading Profile...</Text>
            </View>
        );
    }
```

```
    return (
        <View style={styles.container}>
            <View style={{ backgroundColor: '#FF3F00', paddingVertical: 15, paddingHorizontal: 15, marginTop: 30 }}>
                <Text style={{ color: 'white', fontSize: 20, fontWeight: 'bold' }}>My Profile</Text>
            </View>

            {/* Campos de edición */}
            <View style={styles.container_Inputfield}>
                <FontAwesome5 name="user-alt" size={20} color="#ccc" style={{ paddingLeft: 5, paddingTop: 7 }} />
                <TextInput
                    style={styles.input}
                    placeholder='Full Name'
                    value={editedData.fullName}
                    editable={isEditing}
                    onChangeText={(text) => setEditedData({ ...editedData, fullName: text })}
                />
            </View>

            <View style={styles.container_Inputfield}>
                <Entypo name="email" size={21} color="#ccc" style={{ paddingLeft: 3, paddingTop: 7 }} />
                <TextInput
                    style={styles.input}
                    placeholder='Email'
                    value={editedData.email}
                    editable={false}
                />
            </View>

            <View style={styles.container_Inputfield}>
                <FontAwesome5 name="phone" size={20} color="#ccc" style={{ paddingLeft: 5, paddingTop: 7 }} />
                <TextInput
                    style={styles.input}
                    placeholder='Phone'
                    value={editedData.phone}
                    editable={isEditing}
                    onChangeText={(text) => setEditedData({ ...editedData, phone: text })}
                />
            </View>
```

```
            {/* Botones para alternar entre edición y guardar */}
            {isEditing ? (
                <TouchableOpacity style={styles.button} onPress={handleSave}>
                    <Text style={styles.buttonTxt}>Save</Text>
                </TouchableOpacity>
            ) : (
                <TouchableOpacity style={styles.button} onPress={() => setIsEditing(true)}>
                    <Text style={styles.buttonTxt}>Edit Profile</Text>
                </TouchableOpacity>
            )}
        </View>
    );
};
```

Card Slider Component

The CardSlider.js file is the component that displays available materials organized by categories in the SupplyTrack application. When users reach this screen, they see an attractive and easy-to-navigate visual list divided into specific categories such as "Construction," "Tools," "Finishes," and "plumbing."

This component performs several key tasks:

Uses a horizontal list for each category, displaying materials corresponding to that category.

Each material is presented as a visually clear and organized card.

Each card includes:

An image of the material.

Name of the material.

Supplier of the material.

Price indicated in Mexican Pesos (MXN).

Intelligent Filtering:

Before displaying a category, the component checks if there are any materials in that category. If none are available, it automatically hides the category to avoid showing empty lists.

User Interaction:

Cards function as interactive buttons; tapping one automatically opens the detailed product screen (ProductScreen.js).

This system ensures a smooth user experience when selecting any product of interest.

```
import { StyleSheet, Text, TouchableOpacity, View, Image, FlatList, SafeAreaView } from 'react-native';
import React from 'react';

const categories = ["Construction", "Tools", "Finishes", "Plomeria"];

const CardSlider = ({ navigation, data, categoryRefs }) => {

    const openProductHandler = (item) => {
        navigation.navigate('ProductScreen', item);
    };

    return (
        <View style={styles.container}>
            {categories.map((category) => {
                const filteredData = data.filter(item => item.MaterialType === category);

                if (filteredData.length === 0) return null; // Oculta categorías vacías

                return (
                    <View
                        key={category}
                        ref={ref => { categoryRefs.current[category] = ref; }}
                        style={styles.categoryContainer}
                    >
                        <Text style={styles.categoryHeader}>{category}</Text>
```

```
<View
    key={category}
    ref={ref => { categoryRefs.current[category] = ref; }}
    style={styles.categoryContainer}
>
    <Text style={styles.categoryHeader}>{category}</Text>
    <SafeAreaView>
        <FlatList
            horizontal
            showsHorizontalScrollIndicator={false}
            data={filteredData}
            keyExtractor={(item) => item.id}
            renderItem={({ item }) => (
                <TouchableOpacity key={item.id} style={styles.card} onPress={() => openProductHandler(item)}>
                    <View>
                        <Image source={{ uri: item.MaterialImageUrl }} style={styles.cardimage} />
                    </View>

                    <View style={styles.cardContent}>
                        <Text style={styles.materialName}>{item.MaterialName}</Text>
                        <Text style={styles.supplierText}>Supplier: {item.Supplier}</Text>
                        <Text style={styles.priceText}>Price: ${item.MaterialPrice} MXN</Text>
                    </View>
                </TouchableOpacity>
            )}
        />
    </SafeAreaView>
</View>
```

Categories Component:

The Categories.js component provides users with quick access to each material category. It consists of a horizontal scrollable menu containing interactive category buttons. Each button displays:

- A representative icon/image.
- The name of the category.
- A distinct background color to visually differentiate categories.

User Interaction:

- Users can tap on a category button to immediately filter the displayed materials according to the selected category.
- Enhances navigation and user convenience by allowing quick transitions between categories.

Visual and Functional Consistency:

- Follows the overall aesthetic and intuitive design principles of the SupplyTrack application.
- Provides visual feedback through elevation and subtle animations for an engaging user experience.

```
const Categories = ({ onCategoryPress }) => {
    return (
        <View style={styles.container}>
            <ScrollView horizontal showsHorizontalScrollIndicator={false}>
                <TouchableOpacity style={[styles.box, { backgroundColor: '#ddfbf3' }]} onPress={() => onCategoryPress('Construction')}>
                    <Image source={require('../Images/icon_1.png')} style={styles.image} />
                    <Text style={styles.text}>Construction</Text>
                </TouchableOpacity>
```

```
<TouchableOpacity style={[styles.box, { backgroundColor: '#f5e5ff' }]} onPress={() => onCategoryPress('Tools')}>
    <Image source={require('../Images/icon_2.png')} style={styles.image} />
    <Text style={styles.text}>Tools</Text>
</TouchableOpacity>
```

```
<TouchableOpacity style={[styles.box, { backgroundColor: '#e5f1ff' }]} onPress={() => onCategoryPress('Finishes')}>
    <Image source={require('../Images/icon_3.png')} style={styles.image} />
    <Text style={styles.text}>Finishes</Text>
</TouchableOpacity>
```

```
<TouchableOpacity style={[styles.box, { backgroundColor: '#ebfde5' }]} onPress={() => onCategoryPress('Plomeria')}>
    <Image source={require('../Images/icon_4.png')} style={styles.image} />
    <Text style={styles.text}>Plomeria</Text>
</TouchableOpacity>
```

Headerbar Component

The Headerbar.js component provides users with location information displayed prominently at the top of the SupplyTrack application interface. This component automatically detects and shows the user's current location, enhancing the personalized user experience.

Automatic Location Detection:

- Requests permission from the user to access location data.
- Uses device GPS to fetch current coordinates (latitude and longitude).
- Converts coordinates into readable location names (city or region and country).

Dynamic Location Display:

- Initially displays "Loading..." until location information is retrieved.
- Updates automatically once the location is obtained, clearly showing the user's city and country or region and country.
- Displays informative messages if location access is denied or if an error occurs.

```
const Headerbar = () => {
    const [locationName, setLocationName] = useState("Cargando...");

    useEffect(() => {
        const getLocation = async () => {
            const { status } = await Location.requestForegroundPermissionsAsync();
            if (status !== 'granted') {
                console.log("Permiso de ubicación denegado.");
                return;
            }

            try {
                const location = await Location.getCurrentPositionAsync({});
                const { latitude, longitude } = location.coords;

                const geocode = await Location.reverseGeocodeAsync({
                    latitude,
                    longitude,
                });

                if (geocode.length > 0) {
                    const { city, region, country } = geocode[0];
                    const locationString = city ? `${city}, ${country}` : `${region}, ${country}`;
                    setLocationName(locationString);
                } else {
                    setLocationName("Ubicación no encontrada");
                }
            } catch (error) {
                console.error("Error obteniendo ubicación:", error);
                setLocationName("Error al obtener ubicación");
            }
        };

        getLocation();
    }, []);
```

```
return (
    <View style={styles.container}>
        <TouchableOpacity style={{ flexDirection: 'row' }}>
            <Ionicons name="location-outline" size={28} color="black" style={{ paddingVertical: 6 }} />
            <View style={{ paddingHorizontal: 5 }}>
                <View>
                    <Text style={{ paddingRight: 3, fontSize: 16, fontWeight: '700' }}>Location</Text>
                </View>
                <Text>{locationName}</Text>
            </View>
        </TouchableOpacity>
    </View>
);
```