

TABLE OF CONTENT

|  |  |  |
| --- | --- | --- |
| **SL** | **Contents** | **Page No.** |
| 1 | Table Of Content | 3 |
| 2 | Abstract | 4 |
| 3 | Introduction | 5 |
| 4 | Features | 6 |
| 5 | Used Technologies | 7 |
| 6 | System Architecture And Requirements | 8 |
| 7 | Project Implementation Using Spiral Model | 9 |
| 8 | System Planning | 10 |
| 9 | Methodology | 12 |
| 10 | User Manual | 13 |
| 11 | Project Management | 14 |
| 12 | Database Design | 15 |
| 13 | Code Snippets | 17 |
| 14 | Screenshots-Large Screen | 27 |
| 15 | Screenshots-Mobile Screen | 38 |
| 16 | Future Scope | 39 |
| 17 | Conclusion | 40 |

ABSTRACT

The Ice Cream Inventory System is an Android-based mobile application designed for Swastik Enterprise, an ice cream company, to manage their inventory and shops. The app allows users to add and view shop details, manage inventory, and maintain a comprehensive record of ice cream products. This project involves using Kotlin, Jetpack Compose, and Firebase to develop a smooth, user-friendly interface and real-time data storage.

INTRODUCTION

**Project Name:** Icecream

**Description:** Swastik Enterprise, a leading ice cream company, needed an Android app to streamline their ice cream inventory and shop management. The app will enable store managers to easily add new shops, view ice cream inventory, and track inventory quantities in real-time. This project is being developed as a freelance task for Swastik Enterprise, and Dolphin IT company is coordinating the task.

Project Objectives

* **Inventory Management:** Enable the management of ice cream stock, including adding, updating, and removing products from the inventory.
* **Shop Management:** Allow users to add and view shop details and manage shop-related information.
* **Real-Time Data:** Provide real-time updates on stock levels and shop availability.
* **User-Friendly Interface:** Create an easy-to-use interface for employees and shop owners to interact with the system.
* **Efficient Searching and Filtering:** Implement search and filter functionalities to quickly find products or shop details.
* **Scalable and Maintainable:** Build a system that can be scaled as the business grows, with a strong focus on code quality and maintainability.

FEATURES

* **Shop Management**: Add new shops and view shop details.
* **Inventory Management**: Add ice cream products and track inventory.
* **Real-time Data**: Fetch and display real-time data using Firebase.
* **Search and Filter**: Search shops based on name, owner, and phone number.
* **Shop Selection**: Allow users to select a shop to manage its inventory.

USED TECHNOLOGIES

* **Android Studio**: The primary IDE for development.
* **Kotlin**: The programming language used for the application.
* **Jetpack Compose**: The UI toolkit for building the app's user interface.
* **Retrofit**: Used for communication with external APIs to handle tasks.
* **Coroutines**: For managing asynchronous tasks such as fetching data.
* **Material3**: For UI components like buttons, text fields, and cards.
* **Icons**: Material icons for an intuitive user experience.

SYSTEM ARCHITECTURE AND REQUIREMENTS

**Architecture:**  
The application follows a client-server architecture where the Android app acts as the client, and the external API handles backend operations, including fetching and managing shop and inventory data.

* **Client:** The Android app is built using Kotlin and Jetpack Compose, utilizing modern Android development practices for building the user interface and handling application logic.
* **Backend:** The API provided by Code with Dolphin acts as the backend to manage data tasks such as adding, updating, and retrieving shop and inventory information.
* **UI Design:** The user interface is designed using Material3 and Jetpack Compose, ensuring a modern, responsive, and seamless user experience.

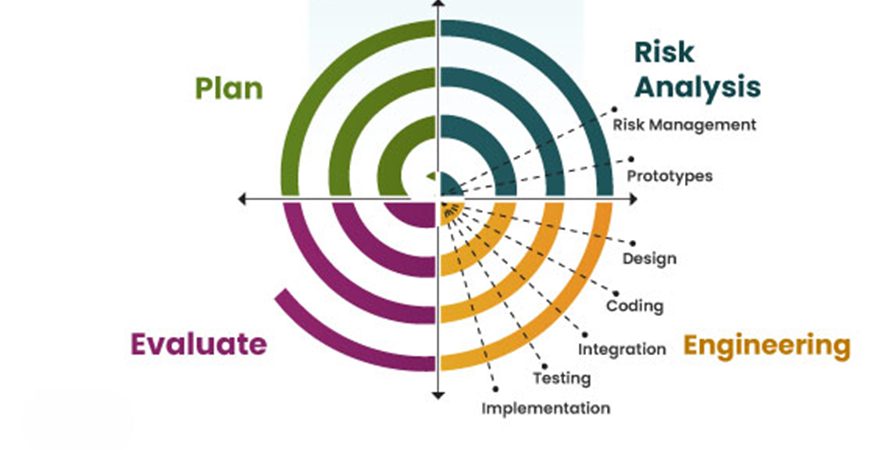
**Hardware Requirements:**

* Android smartphone or tablet running Android 5.0 (Lollipop) or higher.

**Software Requirements:**

* Android Studio with Kotlin support.
* An active Firebase account (for potential integration purposes like authentication or analytics).
* Retrofit library for API communication.

PROJECT IMPLEMENTETION USING SPIRAL MODEL

SPIRAL MODEL Spiral model is one of the most important Software Development Life Cycle models, which provides support for Risk Handling. In its diagrammatic representation, it looks like a spiral with many loops. The exact number of loops of the spiral is unknown and can vary from project to project. Each loop of the spiral is called a Phase of the software development process. The exact number of phases needed to develop the product can be varied by the project manager depending upon the project risks. As the project manager dynamically determines the number of phases, so the project manager has an important role to develop a product using the spiral model. The Spiral Model is a software development life cycle (SDLC) model that provides a systematic and iterative approach to software development. It is based on the idea of a spiral, with each iteration of the spiral representing a complete software development cycle, from requirements gathering and analysis to design, implementation, testing, and maintenance.

SYSTEM PLANNING

System Planning Project Planning:

The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule, These estimates are made within a limited time frame at the beginning of a software project and should be updated regularly as the project progresses, In addition, estimates should attempt to define best case worst case scenarios that project outcomes can be bounded, The planning objective is achieved through a process of information discovery that leads to reasonable estimates.

Project Scheduling:

Project scheduling is an activity that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. The number of basic principles guide the project scheduling is as follows:

* **Compartmentalization:** The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are decomposed.
* **Interdependence:** The interdependence of each compartmentalized activity or task must be determined. Some tasks must occur in-sequence while others can occur in parallel. Some activities cannot commence until the work product produced by another is available. Other activities can occur independently.
* **Time allocation:** Each task is scheduled must be allocated some number of work units. In addition, each task must be assigned a start date and a completion date.
* **Defined outcomes:** Every task that is scheduled should have a defined outcome. For software projects the outcome is normally a work product or a part of a work product.
* **Defined milestones:** Every tasks or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality and has been approved.

Methodology

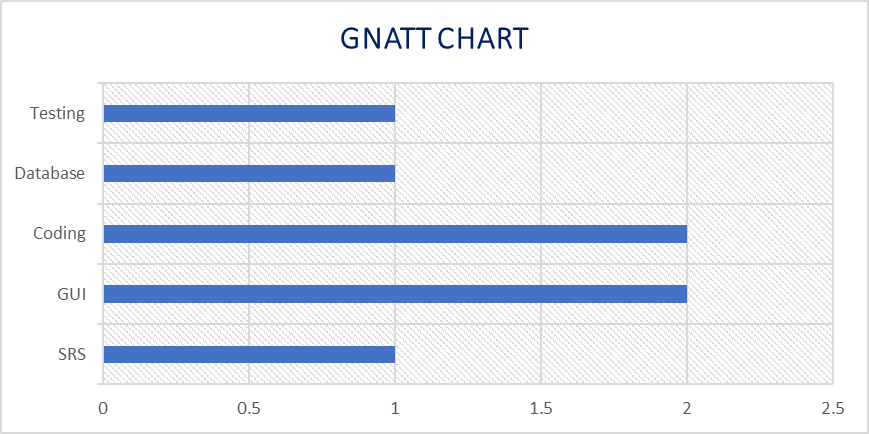
The **Agile methodology** is followed, with iterative development cycles and regular updates. The features are prioritized based on client needs, and the development is adjusted after each phase.

USER MANUAL

* **Launching the App**: Once the app is installed, open the application.
* **Adding a Shop**: Use the "Add Shop" button to add a new shop with details like name, owner, phone number, and location.
* **Managing Inventory**: Click on a shop to add inventory items, view quantities, and manage product details.
* **Searching Shops**: Use the search bar to filter shops based on name, owner, or phone number.

PROJECT MANAGEMENT

Timeline

* **Research and Planning**: 1 days
* **Design**: 2 days
* **Development**: 3 days
* **Testing**: 1 days
* **Deployment**: 1 week

**GNATT CHART**

CODE SNIPPETS

Connection File code:

package itstack.nil.icecream.RetrofitProps  
  
import okhttp3.OkHttpClient  
import okhttp3.logging.HttpLoggingInterceptor  
import retrofit2.Retrofit  
import retrofit2.converter.gson.GsonConverterFactory  
import com.google.gson.GsonBuilder  
  
object RetrofitInstance {  
 private const val BASE\_URL = "https://codeofdolphins.com/"  
  
 fun getInstance(): Retrofit {  
 val interceptor = HttpLoggingInterceptor().*apply* **{** level = HttpLoggingInterceptor.Level.*BODY* **}** val client = OkHttpClient.Builder()  
 .addInterceptor(interceptor)  
 .build()  
  
 val gson = GsonBuilder().setLenient().create()  
  
 return Retrofit.Builder()  
 .baseUrl(BASE\_URL)  
 .addConverterFactory(GsonConverterFactory.create(gson))  
 .client(client)  
 .build()  
 }  
}

Queries Used In Application:

package itstack.nil.icecream.RetrofitProps  
import itstack.nil.icecream.Responses.AllOrderResponse  
import itstack.nil.icecream.Responses.CategoriesResponses  
import itstack.nil.icecream.Responses.GetAllShopResponses  
import itstack.nil.icecream.Responses.LoginResponse  
import retrofit2.Call  
import retrofit2.http.Body  
import retrofit2.http.Field  
import retrofit2.http.FormUrlEncoded  
import retrofit2.http.GET  
import retrofit2.http.Header  
import retrofit2.http.POST  
  
interface ApiInterface {  
 @FormUrlEncoded  
 @POST("inventory/api/login")  
 fun userLogin(  
 @Field("phone") phone: String**,** @Field("password") password: String  
  
 ): Call<LoginResponse>  
  
 @GET("inventory/api/get-all-orders")  
 fun getOrderData(  
 @Header("Authorization") token: String  
 ): Call<AllOrderResponse> // Expecting the root object here  
  
 @GET("inventory/api/get-all-shops")  
 fun getAllShop(  
 @Header("Authorization") token: String  
 ): Call<GetAllShopResponses>  
  
 @GET("inventory/api/get-gategory")  
 fun getAllCategories(  
 @Header("Authorization") token: String  
 ): Call<CategoriesResponses>  
  
 @POST("inventory/api/add-new-shop")  
 fun addNewShop(  
 @Field("shop\_name") shop\_name: String**,** @Field("owner\_name") owner\_name: String**,** @Field("whatsapp\_number") whatsapp\_number: String**,** @Field("address") address: String  
 ): Call<CategoriesResponses>  
}

Login Frontend Code:

package itstack.nil.icecream.LoginScreen  
  
import android.util.Log  
import androidx.compose.foundation.Image  
import androidx.compose.foundation.background  
import androidx.compose.foundation.layout.Column  
import androidx.compose.foundation.layout.Row  
import androidx.compose.foundation.layout.Spacer  
import androidx.compose.foundation.layout.fillMaxSize  
import androidx.compose.foundation.layout.fillMaxWidth  
import androidx.compose.foundation.layout.padding  
import androidx.compose.foundation.layout.width  
import androidx.compose.foundation.shape.RoundedCornerShape  
import androidx.compose.material.icons.Icons  
import androidx.compose.material.icons.automirrored.filled.*Send*import androidx.compose.material.icons.filled.*Person*import androidx.compose.material.icons.filled.*Phone*import androidx.compose.material.icons.filled.*Send*import androidx.compose.material3.Button  
import androidx.compose.material3.ButtonDefaults  
import androidx.compose.material3.Icon  
import androidx.compose.material3.OutlinedTextField  
import androidx.compose.material3.Text  
import androidx.compose.runtime.Composable  
import androidx.compose.ui.Modifier  
import androidx.compose.ui.graphics.Color  
import androidx.compose.ui.res.painterResource  
import androidx.compose.ui.tooling.preview.Preview  
import androidx.compose.ui.unit.dp  
import androidx.compose.runtime.\*  
import itstack.nil.icecream.R  
import itstack.nil.icecream.ui.theme.*LoginBG*import android.widget.Toast  
import androidx.compose.ui.platform.*LocalContext*import androidx.navigation.NavController  
  
@Composable  
fun loginScreen(navController: NavController){  
 var phone by remember **{** *mutableStateOf*("") **}** var password by remember **{** *mutableStateOf*("") **}** val context = *LocalContext*.current  
 Column(  
 **modifier =** Modifier.*fillMaxSize*()  
 ) **{** Row (**modifier =** Modifier.*fillMaxWidth*()  
 .*background*(*LoginBG*)  
 .*weight*(**.6f**))**{** Image(**painter =** painterResource(R.drawable.*icecream*)**,  
 contentDescription =** null**,  
 modifier =** Modifier.*fillMaxSize*()  
 )  
 **}** Row(**modifier =** Modifier.*fillMaxWidth*()  
 .*weight*(**.4f**)) **{** Column(  
 **modifier =** Modifier.*fillMaxSize*()  
 .*padding*(**top = 50**.*dp*)  
 ) **{** OutlinedTextField(  
 **value =** phone**,  
 onValueChange = {** phone = **it },  
 label = {** Text("Phone Number") **},  
 placeholder = {** Text("Enter phone number") **},  
 leadingIcon = {** Icon(Icons.Default.*Phone***, contentDescription =** "Person Icon") **},  
 modifier =** Modifier  
 .*fillMaxWidth*()  
 .*padding*(**horizontal = 8**.*dp*)  
 )  
  
 OutlinedTextField(  
 **value =** password**,  
 onValueChange = {** password = **it },  
 label = {** Text("Password") **},  
 placeholder = {** Text("Enter password") **},  
 leadingIcon = {** Image(**painter =** painterResource(R.drawable.*password*)**, contentDescription =** "Person Icon") **},  
 modifier =** Modifier  
 .*fillMaxWidth*()  
 .*padding*(**horizontal = 8**.*dp*)  
 )  
 Button(  
 **onClick = {** when {  
 phone.*isEmpty*() -> {  
 Toast.makeText(context**,** "Phone number is required!"**,** Toast.*LENGTH\_SHORT*).show()  
 }  
 !phone.*matches*("\\d{10}".*toRegex*()) -> {  
 Toast.makeText(context**,** "Enter a valid 10-digit phone number!"**,** Toast.*LENGTH\_SHORT*).show()  
 }  
 password.*isEmpty*() -> {  
 Toast.makeText(context**,** "Password is required!"**,** Toast.*LENGTH\_SHORT*).show()  
 }  
 password.length < **6** -> {  
 Toast.makeText(context**,** "Password must be at least 6 characters!"**,** Toast.*LENGTH\_SHORT*).show()  
 }  
 else -> {  
 val loginHandler = LoginHandler(phone**,** password**,** context)  
 loginHandler.logMeIn(navController)  
 }  
 }  
 **},  
 colors =** ButtonDefaults.buttonColors(**containerColor =** Color.Blue)**,  
 modifier =** Modifier  
 .*fillMaxWidth*()  
 .*padding*(**vertical = 20**.*dp***, horizontal = 10**.*dp*)**,  
 shape =** *RoundedCornerShape*(**8**.*dp*)  
 ) **{** Icon(Icons.Default.*Send***, contentDescription =** "Send Icon"**, tint =** Color.White)  
 Spacer(**modifier =** Modifier.*width*(**8**.*dp*))  
 Text(**text =** "Login"**, color =** Color.White)  
 **}  
  
  
 }  
 }  
 }**}

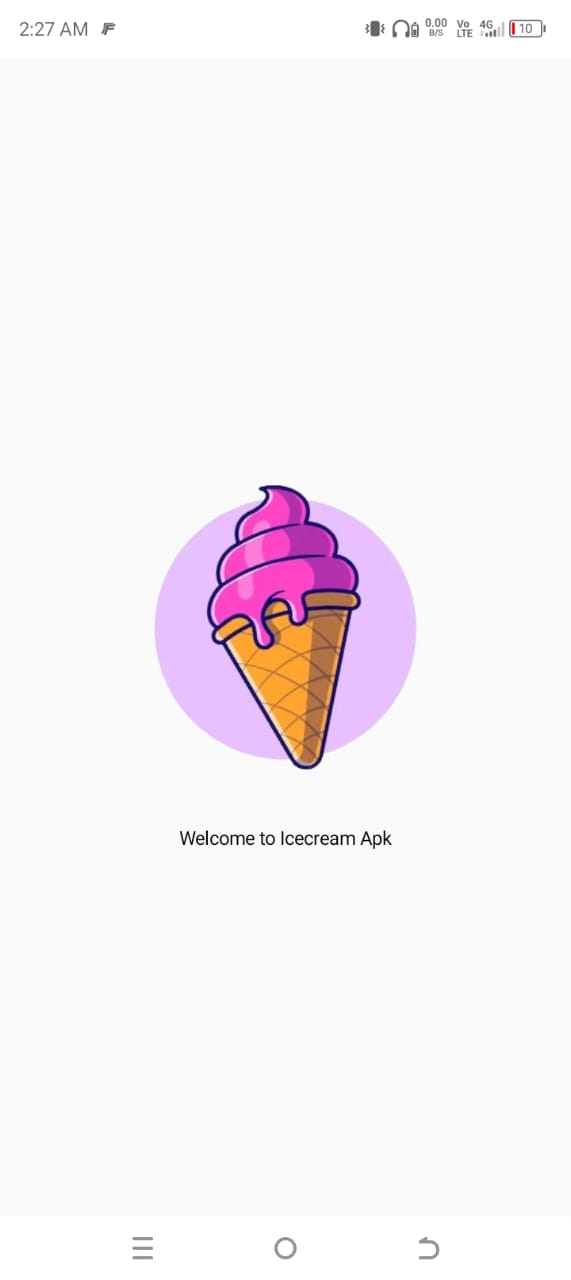
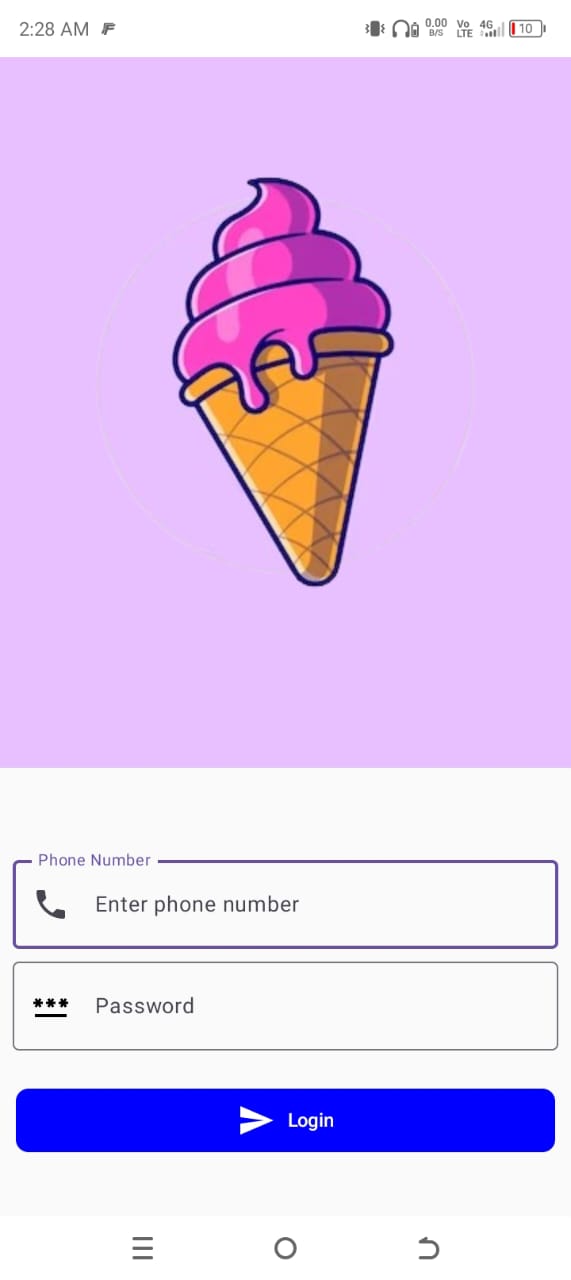
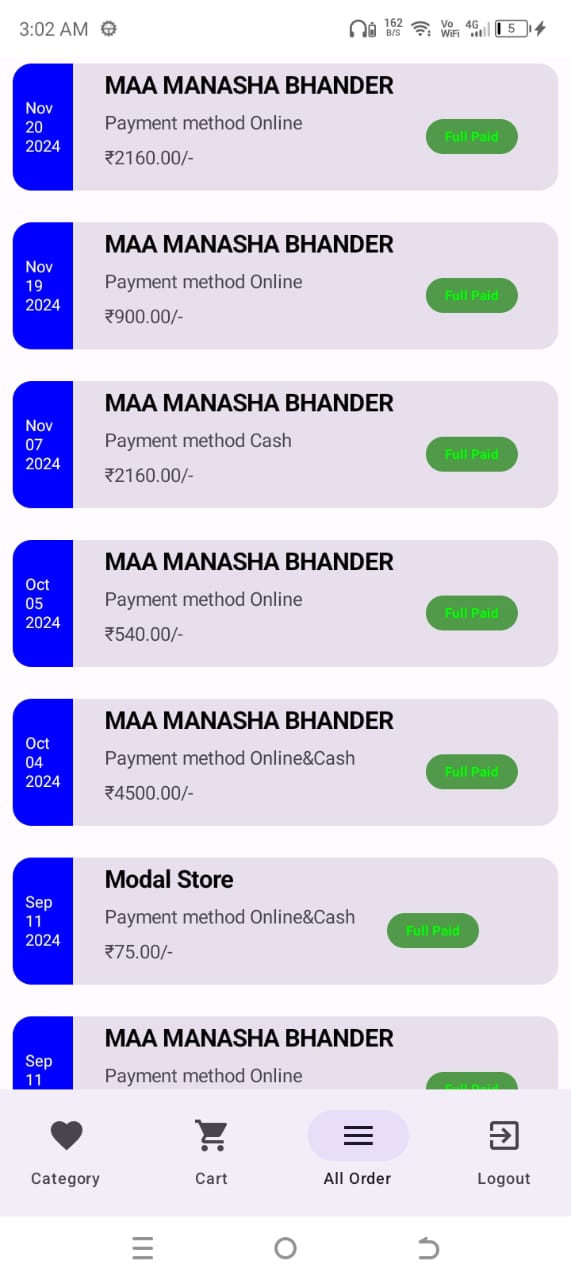
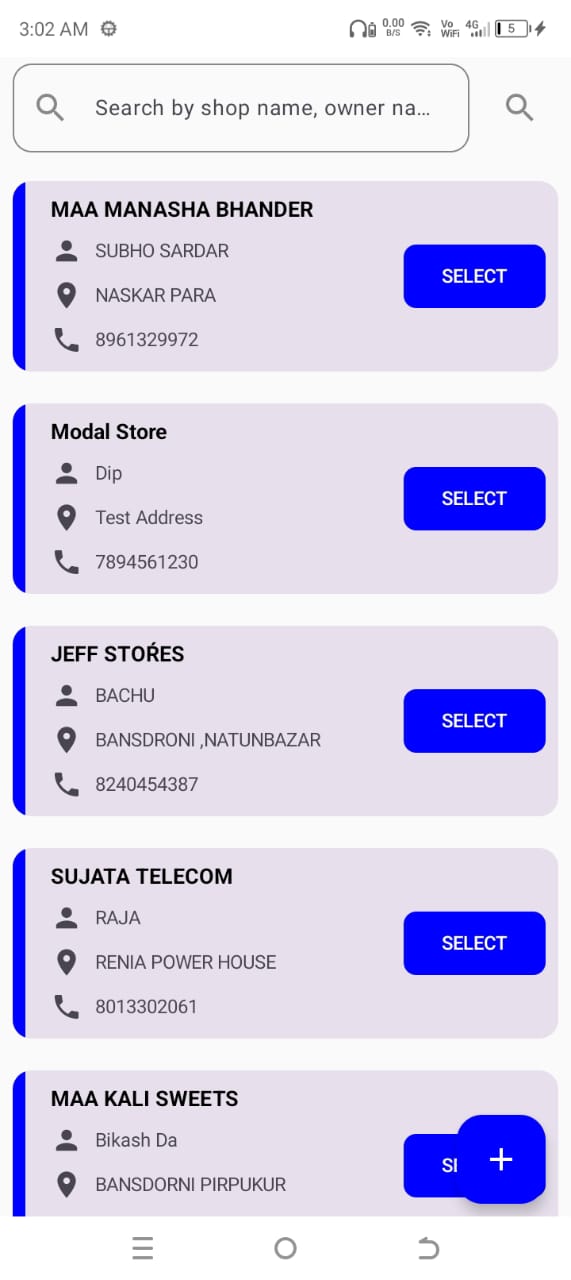
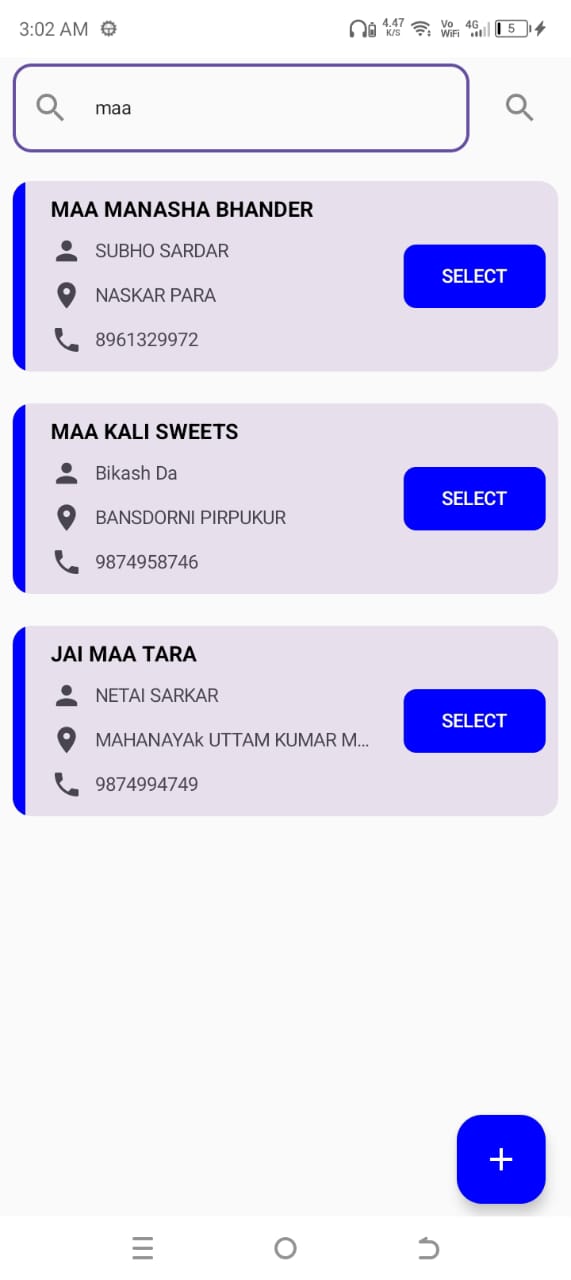
Login Backend Code

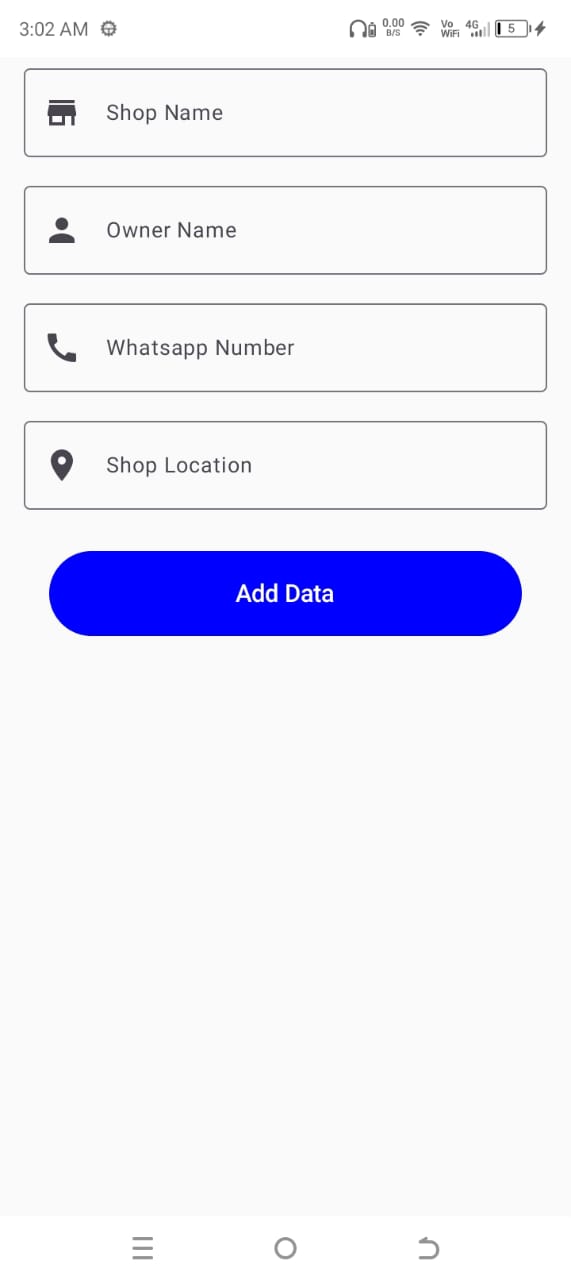
package itstack.nil.icecream.LoginScreen  
  
import android.content.Context  
import android.content.Intent  
import itstack.nil.icecream.RetrofitProps.ApiInterface  
import itstack.nil.icecream.RetrofitProps.RetrofitInstance  
import retrofit2.Call  
import retrofit2.Callback  
import retrofit2.Response  
import android.util.Log  
import android.widget.Toast  
import androidx.navigation.NavController  
import itstack.nil.icecream.Responses.LoginResponse  
  
class LoginHandler(private val phone: String**,** private val password: String**,** private val context: Context) {  
 val sharedPreferences = context.getSharedPreferences("token"**,** Context.*MODE\_PRIVATE*)  
  
 // Login function  
 fun logMeIn(navController: NavController) {  
 val apiService = RetrofitInstance.getInstance().create(ApiInterface::class.*java*)  
 val call = apiService.userLogin(phone**,** password)  
  
 call.enqueue(object : Callback<LoginResponse> {  
 override fun onResponse(call: Call<LoginResponse>**,** response: Response<LoginResponse>) {  
 if (response.*isSuccessful* && response.body() != null) {  
 val token = response.body()?.token  
 if (!token.*isNullOrEmpty*()) {  
 saveTokenToPreferences(token)  
 Toast.makeText(context**,** "Login successful!"**,** Toast.*LENGTH\_SHORT*).show()  
 navController.navigate("app") // navigate to the app screen  
 } else {  
 Toast.makeText(context**,** "Login failed: Token is empty!"**,** Toast.*LENGTH\_SHORT*)  
 .show()  
 }  
 } else {  
 Toast.makeText(  
 context**,** "Login failed: ${response.message()}"**,** Toast.*LENGTH\_SHORT* ).show()  
 }  
 }  
  
 override fun onFailure(call: Call<LoginResponse>**,** t: Throwable) {  
 Log.e("LoginHandler"**,** "Error: ${t.message}")  
 Toast.makeText(context**,** "Login failed: ${t.message}"**,** Toast.*LENGTH\_SHORT*).show()  
 }  
 })  
 }  
  
 // Save the token to shared preferences  
 private fun saveTokenToPreferences(token: String) {  
 sharedPreferences.edit().putString("user\_token"**,** token).apply()  
 }  
}

Logout Code:

package itstack.nil.icecream.LoginScreen  
  
import android.content.Context  
import android.widget.Toast  
import androidx.navigation.NavController  
  
fun logOut(navController: NavController**,** context: Context) {  
 val sharedPreferences = context.getSharedPreferences("token"**,** Context.*MODE\_PRIVATE*)  
  
 sharedPreferences.edit().remove("user\_token").apply()  
  
 Toast.makeText(context**,** "Logged out successfully!"**,** Toast.*LENGTH\_SHORT*).show()  
  
 navController.navigate("login") **{** popUpTo("login") **{** inclusive = true **}** launchSingleTop = true   
 **}**}

SCREENSHOTS





Future Scop

* **Multi-language Support**: Adding language localization for wider user adoption.
* **Cloud Integration**: Storing data on cloud for better scalability.
* **Product Management**: Adding features for managing product sales and stock.
* **Analytics Integration**: Adding analytics to track usage and sales.

Conclusion

The Ice Cream Inventory System successfully meets the requirements of Swastik Enterprise, providing a streamlined way to manage their shops and inventory. The app is easy to use, secure, and scalable, offering real-time updates through Firebase integration. The project was completed successfully within the allocated time frame and offers potential for future enhancements.