**FarmConnect**

*Empowering Agricultural Commerce through Digital Platform*

*Mini Project Report*

*Submitted by*

**N Amal Thomson**

**Reg. No.: AJC22MCA-2065**

*In Partial fulfillment for the Award of the Degree of*

**MASTER OF COMPUTER APPLICATIONS**

**(MCA TWO YEAR)**

**APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY**



**AMAL JYOTHI COLLEGE OF ENGINEERING**

**KANJIRAPPALLY**

[Affiliated to APJ Abdul Kalam Technological University, Kerala. Approved by AICTE, Accredited by NAAC with ‘A’ grade. Koovappally, Kanjirappally, Kottayam, Kerala – 686518]

# 2023-2024

## DEPARTMENT OF COMPUTER APPLICATIONS

### AMAL JYOTHI COLLEGE OF ENGINEERING

**KANJIRAPPALLY**



**CERTIFICATE**

This is to certify that the Project report, “**FARMCONNECT”** is the bona fide work of **N AMAL THOMSON (Regno: AJC22MCA-2065)** in partial fulfillment of the requirements for the award of the Degree of Master of Computer Applications under APJ Abdul Kalam Technological University during the year 2023-24.

**Ms. Meera Rose Mathew Ms. Meera Rose Mathew**

**Internal Guide Coordinator**

**Rev. Fr. Dr. Rubin Thottupurathu Jose**

**Head of the Department**

**DECLARATION**

I hereby declare that the project report **“FARMCONNECT”** is a bona fide work done at Amal Jyothi College of Engineering, towards the partial fulfilment of the requirements for the award of the Master of Computer Applications (MCA) from APJ Abdul Kalam Technological University, during the academic year 2023-2024.

**Date: N AMAL THOMSON**

**KANJIRAPPALLY Reg: AJC22MCA-2065**

# ACKNOWLEDGEMENT

First and foremost, I thank God almighty for his eternal love and protection throughout the project. I take this opportunity to express my gratitude to all who helped me in completing this project successfully. It has been said that gratitude is the memory of the heart. I wish to express my sincere gratitude to our Manager **Rev. Fr. Dr. Mathew Paikatt** and Principal **Dr. Lillykutty Jacob** for providing good faculty for guidance.

I owe a great depth of gratitude towards our Head of the Department **Rev. Fr. Dr. Rubin Thottupurathu Jose** for helping us. I extend my whole hearted thanks to the project coordinator **Ms. Meera Rose Mathew** for her valuable suggestions and for overwhelming concern and guidance from the beginning to the end of the project. I would also express sincere gratitude to my guide **Ms. Meera Rose Mathew** for her inspiration and helping hand.

I thank our beloved teachers for their cooperation and suggestions that helped me throughout the project. I express my thanks to all my friends and classmates for their interest, dedication, and encouragement shown towards the project. I convey my hearty thanks to my family for the moral support, suggestions, and encouragement to make this venture a success.

N AMAL THOMSON

# ABSTRACT

FarmConnect stands as a revolutionary digital platform, transforming traditional agricultural commerce by seamlessly connecting farmers and buyers. Powered by the dynamic combination of Flutter for front-end development and Firebase for back-end management, the platform empowers users to effortlessly browse, select, and purchase the freshest farm produce directly from growers and producers.

This robust platform is designed with key modules, offering comprehensive tools for administrators to efficiently manage user accounts, moderate product listings, facilitate secure transactions, and ensure a smooth, transparent experience for both farmers and buyers. For farmers, FarmConnect is an empowering tool, enabling them to showcase products, receive seasonal cultivation recommendations, efficiently manage inventory, fulfill orders, and access personalized guidance for sustainable farming practices. Simultaneously, buyers benefit from a user-friendly platform facilitating the discovery of a diverse range of products, easy order placement, real-time tracking of deliveries, and direct engagement with farmers for inquiries and product information.

Through its innovative approach and the integration of Flutter and Firebase technologies, FarmConnect emerges as a transformative force in the agricultural sector, facilitating a more efficient and direct connection between producers and consumers.

**CONTENT**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SL. NO** | | **TOPIC** | **PAGE NO** | |
| **1** | | **INTRODUCTION** |  | |
| **1.1** | | **PROJECT OVERVIEW** |  | |
| **1.2** | | **PROJECT SPECIFICATION** |  | |
| **2** | | **SYSTEM STUDY** |  | |
| **2.1** | | **INTRODUCTION** |  | |
| **2.2** | | **EXISTING SYSTEM** |  | |
| **2.3** | | **DRAWBACKS OF EXISTING SYSTEM** |  | |
| **2.4** | | **PROPOSED SYSTEM** |  | |
| **2.5** | | **ADVANTAGES OF PROPOSED SYSTEM** |  | |
| **3** | | **REQUIREMENT ANALYSIS** |  | |
| **3.1** | | **FEASIBILITY STUDY** |  | |
| **3.1.1** | | **ECONOMICAL FEASIBILITY** |  | |
| **3.1.2** | | **TECHNICAL FEASIBILITY** |  | |
| **3.1.3** | | **BEHAVIORAL FEASIBILITY** |  | |
| **3.1.4** | | **FEASIBILITY STUDY QUESTIONNAIRE** |  | |
| **3.2** | | **SYSTEM SPECIFICATION** |  | |
| **3.2.1** | | **HARDWARE SPECIFICATION** |  | |
| **3.2.2** | | **SOFTWARE SPECIFICATION** |  | |
| **3.3** | | **SOFTWARE DESCRIPTION** |  | |
| **3.3.1** | | **FLUTTER** |  | |
| **3.3.2** | | **FIREBASE** |  | |
| **4** | | **SYSTEM DESIGN** |  | |
| **4.1** | | **INTRODUCTION** |  | |
| **4.2** | | **UML DIAGRAM** |  | |
| **4.2.1** | | **USE CASE DIAGRAM** |  | |
| **4.2.2** | | **SEQUENCE DIAGRAM** |  | |
| **4.2.3** | | **STATE CHART DIAGRAM** |  | |
| **4.2.4** | | **ACTIVITY DIAGRAM** |  | |
| **4.2.5** | | **CLASS DIAGRAM** |  | |
| **4.2.6** | | **OBJECT DIAGRAM** |  | |
| **4.2.7** | | **COMPONENT DIAGRAM** |  | |
| **4.2.8** | | **DEPLOYMENT DIAGRAM** |  | |
| **4.3** | | **USER INTERFACE DESIGN USING FIGMA** |  | |
| **4.4** | | **DATABASE DESIGN** |  | |
| **5** | | **SYSTEM TESTING** |  | |
| **5.1** | | **INTRODUCTION** |  | |
| **5.2** | | **TEST PLAN** |  | |
| **5.2.1** | **UNIT TESTING** | |  |
| **5.2.2** | **INTEGRATION TESTING** | |  |
| **5.2.3** | **VALIDATION TESTING** | |  |
| **5.2.4** | **USER ACCEPTANCE TESTING** | |  |
| **5.2.5** | **AUTOMATION TESTING** | |  |
| **5.2.6** | **WIDGET TESTING** | |  |
| **6** | **IMPLEMENTATION** | |  |
| **6.1** | **INTRODUCTION** | |  |
| **6.2** | **IMPLEMENTATION PROCEDURE** | |  |
| **6.2.1** | **USER TRAINING** | |  |
| **6.2.2** | **TRAINING ON APPLICATION SOFTWARE** | |  |
| **6.2.3** | **SYSTEM MAINTENANCE** | |  |
| **6.2.4** | **INSTALLATION** | |  |
| **7** | **CONCLUSION & FUTURE SCOPE** | |  |
| **7.1** | **CONCLUSION** | |  |
| **7.2** | **FUTURE SCOPE** | |  |
| **8** | **BIBLIOGRAPHY** | |  |
| **9** | **APPENDIX** | |  |
| **9.1** | **SAMPLE CODE** | |  |
| **9.2** | **SCREEN SHOTS** | |  |

## List of Abbreviation

**UML** Unified Modelling Language

**IDE** Integrated Development Environment

**UI** User Interface

**BaaS** Backend-as-a-Service

**SDK** Software Development Kit

# CHAPTER 1

# INTRODUCTION

### PROJECT OVERVIEW

In the ever-changing world of agriculture, FarmConnect has emerged as a revolutionary solution that connects farmers and buyers directly through a sophisticated digital platform. Built with care and powered by Flutter for an intuitive user experience and Firebase for a strong back-end infrastructure, FarmConnect goes beyond traditional boundaries to offer a transformative journey for everyone involved in agriculture.

The agricultural industry has long been characterized by complex supply chains and middlemen, creating inefficiencies that hurt both farmers and consumers. FarmConnect enters this arena as a catalyst for change, using cutting-edge technologies to redefine how fresh farm products are grown, shown, and finally delivered to consumers. This digital ecosystem envisions a future where transparency, efficiency, and sustainability are the cornerstones of agricultural commerce.

FarmConnect is more than just a marketplace; it's a dynamic hub that empowers farmers to expand their reach, adopt sustainable practices, and thrive in the digital age. At the same time, it gives buyers an easy-to-use interface, offering a wide variety of farm products while fostering direct connections with growers. Let's explore the details of FarmConnect's key modules and features that work together to reshape the agricultural landscape.

### PROJECT SPECIFICATION

FarmConnect, an innovative digital platform powered by Flutter for the front-end and Firebase for the back-end, is a transformative force in the realm of agricultural commerce. By seamlessly connecting farmers and buyers, FarmConnect reshapes traditional supply chains, facilitating the direct purchase of fresh farm products. Let's delve into the key modules and features that make FarmConnect a comprehensive solution for both farmers and buyers.

* **Admin**
* Authentication (Email/Google)
* User Management (Enable/Disable)
* Dashboard
* Approve Farmers
* Product Approval
* Email Notification
* Facilitate Payment
* **Farmer**
* Authentication (Email/Google)
* Profile Management (Reset Password/Update Profile)
* Dashboard
* Add Products
* Stock Management
* Facilitate Payment
* Bill Generation
* Email Notification
* **Buyers**
* Authentication (Email/Google)
* Profile Management (Reset Password/Update Profile)
* Dashboard
* Product Search
* Order Placement
* Make Payment
* Download Bills

# CHAPTER 2

# SYSTEM STUDY

# 

### INTRODUCTION

Data collection and analysis, problem-solving, and system change recommendations are all steps in the process of system analysis. During this problem-solving process, there must be considerable communication between the system users and the system developers. A system analysis or research should be the first step in any system development process. The system analyst acts as an interrogator and examines the operation of the current system in great detail. The system's input is acknowledged, and the system is viewed as a whole. The many processes might be connected to the organizations’ outcomes. System analysis involves comprehending the problem, identifying the significant and crucial variables, analyzing and synthesizing the numerous components, and choosing the best or, at the very least, most acceptable course of action.

Preliminary research is the process of gathering and analyzing data in order to use it for upcoming system investigations. Initial research requires strong collaboration between system users and developers since it involves problem-solving. It carries out several feasibility studies. These studies offer a rough idea of the system activities, which can be utilized to choose the methods to employ for effective system research and analysis.

### EXISTING SYSTEM

The existing agricultural commerce system is characterized by a traditional, manual approach that heavily relies on intermediaries and lacks technological integration. Farmers, as primary producers, face challenges in reaching a wider market due to limited visibility and accessibility. Transactions are often opaque, leading to mistrust between buyers and sellers. Manual record-keeping processes contribute to delays and errors in order fulfillment, and communication between farmers and buyers is often inefficient. The dependency on intermediaries increases the overall cost of agricultural products for buyers and reduces profit margins for farmers.

**2.2.1 NATURAL SYSTEM STUDIED**

The natural system studied involves a comprehensive examination of the agricultural ecosystem, considering the seasonal and geographical aspects of crop cultivation, the inherent challenges faced by farmers, and the flow of produce through various stages of the supply chain. By understanding the natural system, including the interactions between farmers, buyers, and intermediaries, we gain insights into the complexities and opportunities within the agricultural sector.

**2.2.2 DESIGNED SYSTEM STUDIED**

The designed system, FarmConnect, introduces a paradigm shift in agricultural commerce. Utilizing the Flutter framework for the front-end and Firebase for the back-end, this digital platform seamlessly connects farmers and buyers. The system is equipped with specialized modules for administrators, farmers, and buyers, each tailored to their specific needs. Key features include a direct connection between farmers and buyers, real-time order tracking, and personalized recommendations based on data-driven insights. Advanced functionalities, such as image recognition for product search, disease detection, and quality assessment tools, enhance the overall user experience.

### DRAWBACKS OF EXISTING SYSTEM

* Limited Market Reach
* Opaque Transactions
* Inefficient Communication
* Manual Processes
* Dependency on Intermediaries:

### PROPOSED SYSTEM

The proposed system, FarmConnect, emerges as a groundbreaking solution to revolutionize the landscape of agricultural commerce. Designed with precision and driven by advanced technologies such as Flutter for the front-end and Firebase for the back-end, FarmConnect aims to bridge the existing gaps in the traditional supply chain. The core of the system lies in fostering a direct connection between farmers and buyers, eliminating the need for intermediaries. This not only enhances transparency in transactions but also establishes a foundation of trust between producers and consumers. The platform's user-friendly interface provides a seamless experience for both farmers and buyers, empowering farmers to showcase their products effectively and buyers to explore a diverse range of fresh farm products. Real-time order tracking and secure payment processes ensure the efficiency of transactions, while advanced features such as image recognition for product search, disease detection, and quality assessment tools add a layer of sophistication to the user experience. FarmConnect goes beyond being a mere marketplace; it aspires to be a digital ecosystem that promotes sustainable farming practices, expands market reach for farmers, and sets new standards in the realm of agricultural commerce.

### ADVANTAGES OF PROPOSED SYSTEM

* Direct Farmer-Buyer Connection
* Efficient Transactions
* Enhanced User Experience
* Transparency and Trust
* Market Expansion for Farmers
* Simplified Administration
* Time Saving

# CHAPTER 3

# REQUIREMENT ANALYSIS

## FEASIBILITY STUDY

A feasibility study is conducted to determine if the project will, upon completion, fulfil the objectives of the organization in relation to the labor, effort, and time invested in it. A feasibility study enables the developer to predict the project's usefulness and potential future. A system proposal's workability, which includes the influence on the organization, capacity to satisfy user demands, and efficient use of resources, is the basis for a feasibility study. As a result, a feasibility analysis is frequently performed before a new application is approved for development. The paper outlines the project's viability and contains a number of factors that were carefully taken into account throughout this project's feasibility assessment, including its technical, economic, and operational viabilities. The following are its features: -

### Economical Feasibility

Economically, FarmConnect demonstrates its viability through a well-structured financial analysis. Initial startup costs, encompassing development expenses, marketing efforts, operational setup, and administration, have been carefully estimated. Revenue projections are based on revenue models that include transaction fees, subscription plans, and other income streams, with income projections taking into account user adoption rates. Operating expenses, such as server hosting, maintenance, marketing, and customer support, have been forecasted to ensure sustainable operations. The project's break-even point, where total revenue equals total costs, has been identified, and sensitivity analysis has been conducted to understand how changes in key assumptions may impact financial viability. Potential funding sources, including investment, loans, grants, and partnerships, have been explored to secure the necessary capital for development and initial operations.

### Technical Feasibility

FarmConnect's technical feasibility is underpinned by a robust technology stack. The selection of Flutter for cross-platform mobile app development and Firebase for backend services offers a strong foundation. Flutter ensures compatibility across both Android and iOS platforms, while Firebase provides essential features such as real-time database management and secure user authentication. The presence of a skilled development team well-versed in these technologies ensures efficient implementation. Moreover, scalable technical infrastructure, including servers, databases, and hosting services, is available to accommodate potential platform growth. Security measures, including data encryption, secure user authentication, and robust protection against data breaches, are in place to safeguard user information and transactions. The platform seamlessly integrates with third-party services, APIs, and data sources, enhancing the user experience. Plans for ongoing maintenance, updates, and technical support for users and administrators are well-defined, ensuring the platform's continued functionality and reliability.

### Behavioral Feasibility

Behaviorally, FarmConnect has been carefully designed to align with user preferences and practices. User acceptance has been assessed to ensure that the platform resonates with both farmers and buyers. User-friendliness and intuitiveness have been emphasized in the platform's design to provide a positive user experience. Continuous user feedback mechanisms are in place to understand user needs, expectations, and concerns, allowing for iterative improvements. The platform actively promotes desirable behavioral changes, such as supporting sustainable farming practices and fostering direct farmer-buyer interactions. User behaviors have been analyzed in the context of competition and market dynamics to adapt to changing conditions.

**3.1.4 Feasibility Study Questionnaire**

**1. Project Overview:**

FarmConnect represents a transformative digital platform poised to revolutionize the agricultural commerce landscape. Developed using the versatile Flutter framework for its front-end and the robust Firebase backend, this platform redefines how farmers and buyers interact in the agricultural sector. Its core objective is to seamlessly connect farmers, who are the primary producers of agricultural products, with buyers who seek fresh, locally sourced goods. FarmConnect strives to empower farmers by expanding their market reach and offers buyers a convenient, transparent, and direct way to access agricultural products.

**2. To what extent the system is proposed for?**

FarmConnect is an extensive system designed to encompass a broad spectrum of functionalities. It serves as an end-to-end solution for agricultural commerce, integrating tools and features that enhance the entire supply chain. From user management to product listings, order processing, and communication capabilities, it offers a comprehensive set of services. These include facilitating secure transactions, offering personalized recommendations based on user preferences, and promoting sustainable farming practices. The system's reach extends across the entire agriculture sector, from small-scale farmers to larger agricultural enterprises.

**3. Specify the Viewers/Public which is to be involved in the System:**

FarmConnect's primary audience comprises three key user groups:

* **Farmers**: These are the backbone of the agricultural sector, and FarmConnect empowers them by providing a platform to showcase their products, efficiently manage inventory, and receive valuable seasonal cultivation recommendations. This not only enhances their market access but also supports sustainable farming practices.
* **Buyers:** Buyers use FarmConnect to access a diverse range of agricultural products. They can easily discover products, place orders, track deliveries in real-time, and engage directly with farmers for inquiries or additional product information. This direct communication fosters trust and transparency in the agricultural supply chain.
* **Admins:** The administrative team ensures the platform's smooth operation. They oversee user accounts, moderate product listings, provide support to users, and manage the overall functionality of the platform. Admins play a crucial role in maintaining the platform's integrity and efficiency.

**4. List the Modules included in your System:**

FarmConnect consists of three core modules:

* **Admin Module:** This module equips administrators with comprehensive tools to manage user accounts, review and approve product listings, facilitate secure transactions, and provide robust support to farmers and buyers.
* **Farmer Module:** Empowering farmers to present their products, efficiently manage inventory and orders, receive personalized cultivation recommendations, and engage in direct communication with both buyers and administrators.
* **Buyer Module:** Providing buyers with an intuitive interface for product discovery, order placement, real-time delivery tracking, and direct communication with farmers for inquiries and product details.

**5. Identify the users in your project:**

* **Admins:** Responsible for overseeing the entire platform, admins ensure smooth operations, manage user accounts, moderate product listings, and provide support to users.
* **Farmers:** These primary producers use FarmConnect to showcase their products, manage orders and inventory, receive valuable cultivation recommendations, and communicate directly with buyers and admins.
* **Buyers:** Buyers leverage FarmConnect for discovering, selecting, and purchasing agricultural products, with features including real-time order tracking, communication with farmers, and access to their order history.

**6. Who owns the system?**

The ownership details typically depend on the organization or individuals who initiated and funded the development of the platform.

* Entrepreneur or Founder
* Start-up Company
* Agricultural Cooperative
* Non-Profit Organization
* Government or Public Entity

**7. System is related to which firm/industry/organization?**

FarmConnect's direct connection is with the agricultural industry. It serves as a catalyst for improving the efficiency, transparency, and sustainability of agricultural commerce. By connecting farmers and buyers directly, FarmConnect contributes to the growth and prosperity of local agricultural ecosystems and empowers farmers to access broader markets. It aligns with the goals of agricultural organizations, government bodies, and industry associations committed to supporting sustainable farming practices and enhancing market access for growers.

**8. Details of the person that you have contacted for data collection:**

1. Mostly Online Sources
2. The process of data collection for FarmConnect involves collaboration with a diverse range of stakeholders. It encompasses collecting and curating essential data, such as product information, user profiles, and regional agricultural data. This data collection effort requires engagement with multiple parties, including:

* Farmers and Agricultural Experts
* Government Agencies
* Farmers' Associations
* Local Agricultural Communities.

**9. Questionnaire to collect details about the project:**

1. **What types of products do you list or purchase on FarmConnect?**

I do not list or purchase products on the platform myself. My role primarily involves managing user accounts, monitoring platform activities, and facilitating payment processing.

1. **Are there specific features or functionalities you would like to see added to FarmConnect?**

Adding additional features like real-time chat support for users, more comprehensive reporting and analytics tools, and tools for promoting sustainable farming practices would be valuable enhancements.

1. **Provide more information about your target user base? Are there any specific demographics or regions you are primarily focusing on?**

FarmConnect aims to cater to a diverse user base that includes:

Farmers: Both small-scale and large-scale farmers who want to expand their market reach.

Buyers: Individuals, restaurants, and businesses looking for quality farm products.

1. **Are there any unique features or functionalities you envision that will set FarmConnect apart from other agricultural commerce platforms?**

To set FarmConnect apart from other agricultural commerce platforms, we envision unique features such as:

* Personalized cultivation recommendations based on local weather data.
* Real-time order tracking for buyers.
* Direct communication channels between farmers and buyers.
* Robust image recognition for product search.
* Integrated disease detection and pest control guidance.

1. **How do you plan to ensure the security of user data and financial transactions on the platform?**

To ensure the security of user data and financial transactions, FarmConnect will implement:

* Robust encryption protocols.
* Regular security audits.
* Compliance with industry best practices.

1. **Are there any specific technologies or integrations you require for the image recognition, disease detection, and pest control modules?**

We plan to integrate machine learning models and image processing technologies for image recognition, disease detection, and pest control modules. Collaborations with specialized technology providers will be pursued.

1. **How do you plan to gather and maintain information on organic products, and how will they be distinguished on the platform?**

Information on organic products will be collected through a separate section on the platform. They will be distinguished by relevant certifications and labels, ensuring transparency for buyers.

1. **What specific functionalities should the admin dashboard include?**

The admin dashboard in the FarmConnect platform should provide comprehensive tools and functionalities to enable administrators to effectively manage user accounts, oversee platform operations, and ensure a smooth and secure experience for all users.

* User Management
* Product Approval
* Reports
* Facilitate Payment
* Manage Bills and Orders

1. **Are there any legal or regulatory requirements that the platform must adhere to, especially in the agricultural sector?**

Yes, there may be legal and regulatory requirements that the FarmConnect platform must adhere to, especially in the agricultural sector. Compliance with these regulations is essential to ensure the platform operates within the bounds of the law and maintains the trust of users. The specific regulations and requirements can vary by region and country, but some common considerations in the agricultural sector include:

* Data Privacy and Protection
* Food Safety and Quality Standards
* Organic Certification
* Pesticide and Chemical Regulations
* Intellectual Property

1. **What types of notifications and alerts should users receive?**

Users of the FarmConnect platform should receive various types of notifications and alerts to keep them informed about important events, updates, and interactions on the platform. These notifications are essential for providing a smooth and engaging user experience

* Farmer Approval
* Enable/Disable Updates
* Order Updates
* Payment Updates
* Product Updates

## 3.2 SYSTEM SPECIFICATION

### Hardware Specification

Device Processor - Apple iPhone – Apple Bionic & Android - Qualcomm

Snapdragon, Samsung Exynos

Device RAM - 4 GB or more

Device Storage - 64 GB or more

### Software Specification

Front End - Dart, Flutter

Back End - Firebase

Database - Firebase Firestore

Client on Device - iOS10 and above or Android 7 or above

Technologies used - Dart, Flutter, RazorPay, Firebase Firestore, Firebase Storage

& Firebase APIs

## SOFTWARE DESCRIPTION

### Flutter

In May 2017, Google unveiled Flutter, a revolutionary free and open-source mobile UI framework.

In essence, Flutter empowers developers to craft native mobile applications using a unified code-

base. This breakthrough means that you can develop distinct applications for both iOS and Android

platforms, all within the same programming language and code-base.

Flutter consists of two important parts:

* **SDK (Software Development Kit):** This toolkit equips you with a set of tools to create applications. It also provides the means to translate your code into native machine code, tailored for both iOS and Android platforms.
* **Framework (UI Library with Widgets):** This encompasses a versatile collection of reusable user interface (UI) elements that you can customize to align with your specific requirements. These elements encompass buttons, text input fields, sliders, and various other interactive objects.

Applications developed with Flutter utilize the Dart programming language. Although Google introduced Dart in October 2011, it has undergone significant advancements since its inception. Dart serves as a front-end coding framework that empowers the creation of programs for both web and mobile platforms.

### Firebase

Firebase is a backend-as-a-service (Baas). It gives developers a variety of tools and services so they can produce high-quality apps, increase their user base, and earn money. It was developed on the technology platform of Google. Data is stored in documents that resemble JSON in Firebase, a NoSQL database program.

Key features of firebase:

* **Authentication**: Firebase supports authentication. Passwords, facebook, google, phone number etc can be used to authenticate. We can even integrate more than one sign-in method into our app using firebase.
* **Realtime Database:** Data is available and synced across all the clients in realtime, even if and when the app goes offline.
* **Hosting:** Firebase offers the app quick hosting.
* **Test lab:** The app can be tested on physical devices located on Google’s data center or even on virtual devices.
* **Performance Monitoring:** This feature helps developers gain insights into the performance of their app, allowing them to optimize it for better user experiences.
* **Cloud Firestore:** This is a flexible, scalable No-SQL cloud database that allows for seamless data storage and synchronization in real-time.
* **Cloud Messaging:** Firebase Cloud Messaging (FCM) enables reliable and efficient delivery of messages to target devices, including iOS, Android, and web.
* **Crashlytics:** Firebase Crashlytics provides detailed crash reports, allowing developers to quickly identify and fix issues in their apps.

# CHAPTER 4

# SYSTEM DESIGN

* 1. **INTRODUCTION**

System design is a critical phase in the process of application development, playing a pivotal role in shaping the architecture and functionality of software. It encompasses the creation of a structured blueprint that outlines how various components, modules, and services within the application will interact and collaborate to meet specific objectives. This process involves making crucial decisions regarding database architecture, server configuration, and overall system architecture to ensure scalability, efficiency, and robustness. Additionally, system design takes into account factors such as user experience, security, and data management to create a well-rounded and effective application. It requires a deep understanding of the application's requirements, as well as proficiency in various technologies and programming paradigms to construct a robust foundation for the development process.

In essence, system design serves as the architectural backbone of any software project. It involves

breaking down the complex functionalities and requirements of the application into manageable components, each with a defined purpose and relationship with other elements. Through meticulous planning and consideration of various technical and user-centric aspects, system design lays the groundwork for developers to implement code efficiently and cohesively. A well-crafted system design not only ensures that the application meets performance and scalability requirements but also provides a framework for future enhancements and maintenance. It is a critical step that bridges the gap between conceptualizing an application and transforming it into a functional, reliable, and user-friendly software solution.

## UML DIAGRAM

Unified Modeling Language (UML) stands as a foundational tool in software engineering, renowned for its role in visually representing complex systems and processes. It provides a standardized set of graphical notations that facilitate the clear depiction of various aspects of a system's structure and behavior. Originating from the collaboration of industry experts, UML has gained widespread acceptance and adoption in both academia and industry. It serves as a powerful communication tool, enabling stakeholders, including developers, designers, and clients, to attain a shared understanding of system architecture, design, and functionality. UML diagrams act as a lingua franca, transcending language barriers and ensuring a consistent means of conveying intricate software concepts, ultimately enhancing the efficiency and effectiveness of the software development process.

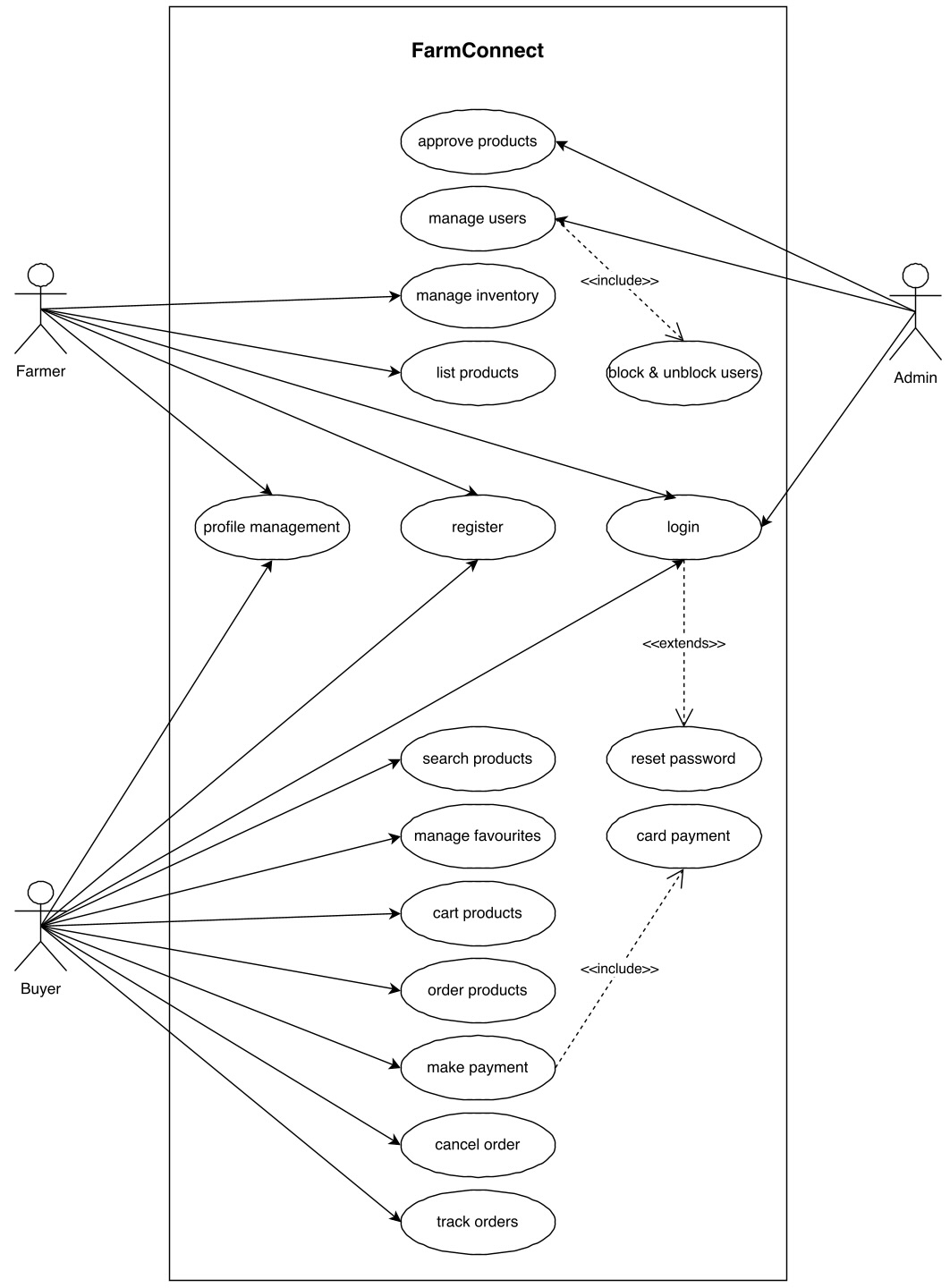
**Types of UML diagrams**

* Class diagram
* Object diagram
* Use case diagram
* Sequence diagram
* Activity diagram
* State chart diagram
* Deployment diagram
* Component diagram

## Use Case Diagram

Use Case Diagrams, a cornerstone in software engineering, serve as a visual representation of the interactions between a system and its external entities. At their core, they provide a structured means of identifying and defining the various functionalities a system offers and how these functionalities are accessed by different actors or entities. Actors, representing users, systems, or external entities, are depicted along with the specific use cases they engage with. Associations between actors and use cases elucidate the nature of these interactions, clarifying the roles and responsibilities of each entity within the system. This detailed visual representation not only enhances communication among stakeholders but also provides a clear blueprint for system functionality, laying the foundation for the subsequent stages of the software development process. Overall, Use Case Diagrams play a pivotal role in aligning development efforts with user expectations, ensuring that the resulting software system fulfills its intended purpose effectively and efficiently.

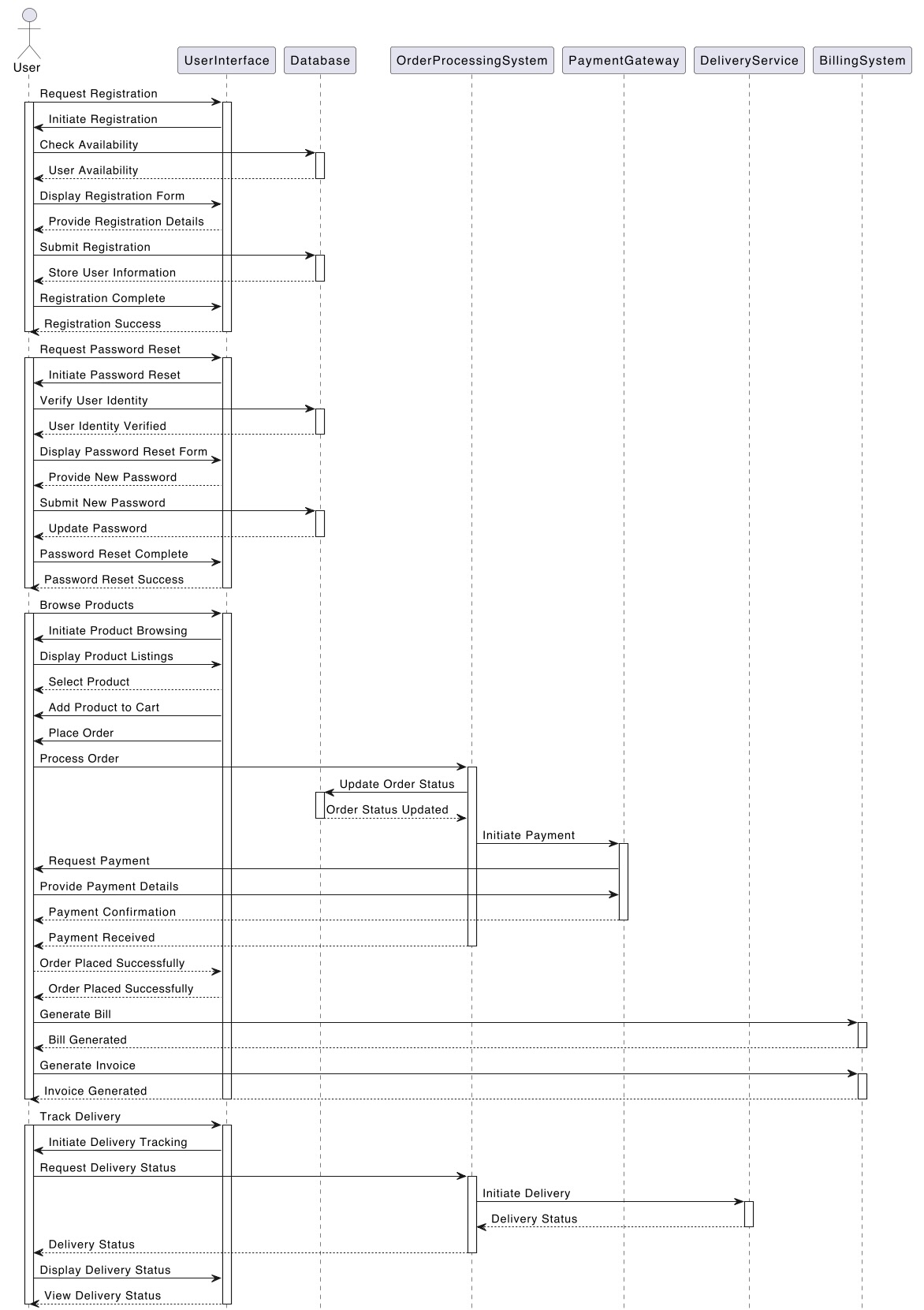
* **Actor Definition:** Clearly define and label all actors involved in the system. Actors represent external entities interacting with the system.
* **Use Case Naming:** Use descriptive names for use cases to accurately convey the functionality they represent.
* **Association Lines:** Use solid lines to represent associations between actors and use cases. This signifies the interaction between entities.
* **System Boundary:** Draw a box around the system to indicate its scope and boundaries. This defines what is inside the system and what is outside.
* **Include and Extend Relationships:** Use "include" relationships to represent common functionalities shared among multiple use cases. Use "extend" relationships to show optional or extended functionalities.

****

***Use Case Diagram***

## 4.2.2 Sequence Diagram

Sequence Diagrams stand as dynamic models in software engineering, portraying the chronological flow of interactions between various objects or components within a system. They spotlight the order in which messages are exchanged, revealing the behavior of the system over time. Actors and objects are represented along a vertical axis, with arrows indicating the sequence of messages and their direction. Lifelines, extending vertically from actors or objects, illustrate their existence over the duration of the interaction. These diagrams serve as a vital tool for visualizing system behavior and understanding the temporal aspects of a software process. Through Sequence Diagrams, stakeholders gain valuable insights into how different elements collaborate to achieve specific functionalities, facilitating more effective communication among development teams and stakeholders alike. This detailed representation not only aids in detecting potential bottlenecks or inefficiencies but also provides a foundation for refining system performance in the later stages of software development.

* **Vertical Ordering:** Represent actors and objects along a vertical axis, indicating the order of interactions from top to bottom.
* **Lifelines:** Extend vertical lines from actors or objects to denote their existence and participation in the interaction.
* **Activation Bars:** Use horizontal bars along lifelines to show the period during which an object is active and processing a message.
* **Messages and Arrows:** Use arrows to indicate the flow of messages between objects, specifying the direction of communication.
* **Self-Invocation:** Use a looped arrow to represent self-invocation, when an object sends a message to itself.
* **Return Messages:** Indicate return messages with a dashed arrow, showing the response from the recipient.
* **Focus on Interaction:** Sequence Diagrams focus on the chronological order of interactions, avoiding implementation details.
* **Concise Notation:** Use clear and concise notation to represent messages and interactions, avoiding unnecessary complexity.
* **Consider System Boundaries:** Clearly define the boundaries of the system to indicate what is included in the interaction.
* **Feedback and Validation:** Seek feedback from stakeholders and team members to ensure the diagram accurately represents the system behavior.

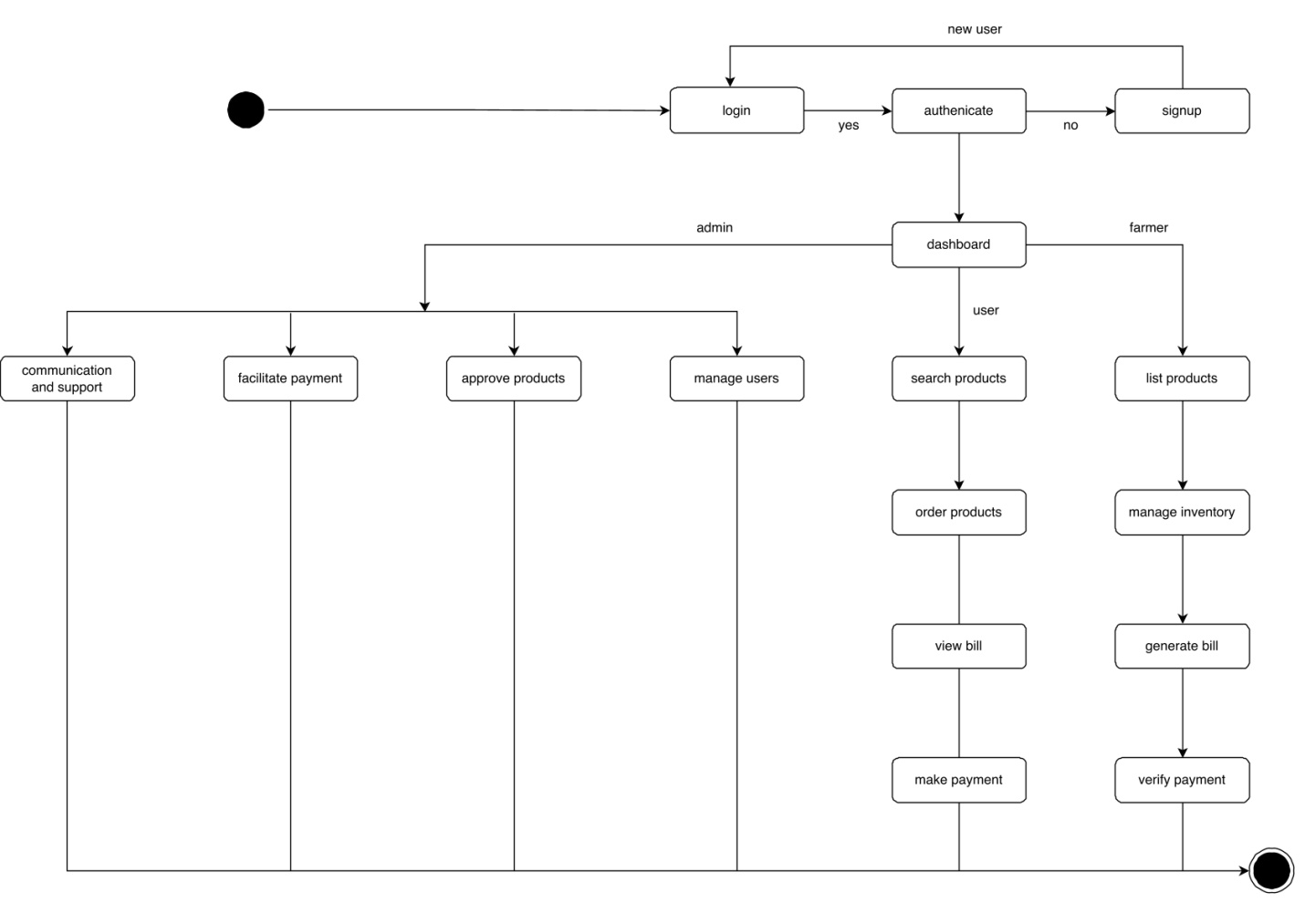
***Sequence Diagram***

## 4.2.3 State Chart Diagram

A State Chart Diagram, a fundamental component of UML, provides a visual representation of an object's lifecycle states and the transitions between them. It depicts the dynamic behavior of an entity in response to events, showcasing how it transitions from one state to another. Each state represents a distinct phase in the object's existence, while transitions illustrate the conditions triggering state changes. Initial and final states mark the commencement and termination of the object's lifecycle. Orthogonal regions allow for concurrent states, capturing multiple aspects of the object's behavior simultaneously. Hierarchical states enable the representation of complex behaviors in a structured manner. Entry and exit actions depict activities occurring upon entering or leaving a state. Moreover, guard conditions ensure that transitions occur only under specified circumstances. State Chart Diagrams play a crucial role in understanding and designing the dynamic behavior of systems, aiding in the development of robust and responsive software applications.

**Key notations for State Chart Diagrams:**

* **Initial State:** Represented by a filled circle, it signifies the starting point of the object's lifecycle.
* **State:** Depicted by rounded rectangles, states represent distinct phases in an object's existence.
* **Transition Arrow:** Arrows denote transitions between states, indicating the conditions triggering a change.
* **Event:** Events, triggers for state changes, are labeled on transition arrows.
* **Guard Condition:** Shown in square brackets, guard conditions specify criteria for a transition to occur.
* **Final State:** Represented by a circle within a larger circle, it indicates the end of the object's lifecycle.
* **Concurrent State:** Represented by parallel lines within a state, it signifies concurrent behaviors.
* **Hierarchy:** States can be nested within other states to represent complex behavior.
* **Entry and Exit Actions:** Actions occurring upon entering or leaving a state are labeled within the state.
* **Transition Labels:** Labels on transition arrows may indicate actions or operations that accompany the transition.

****

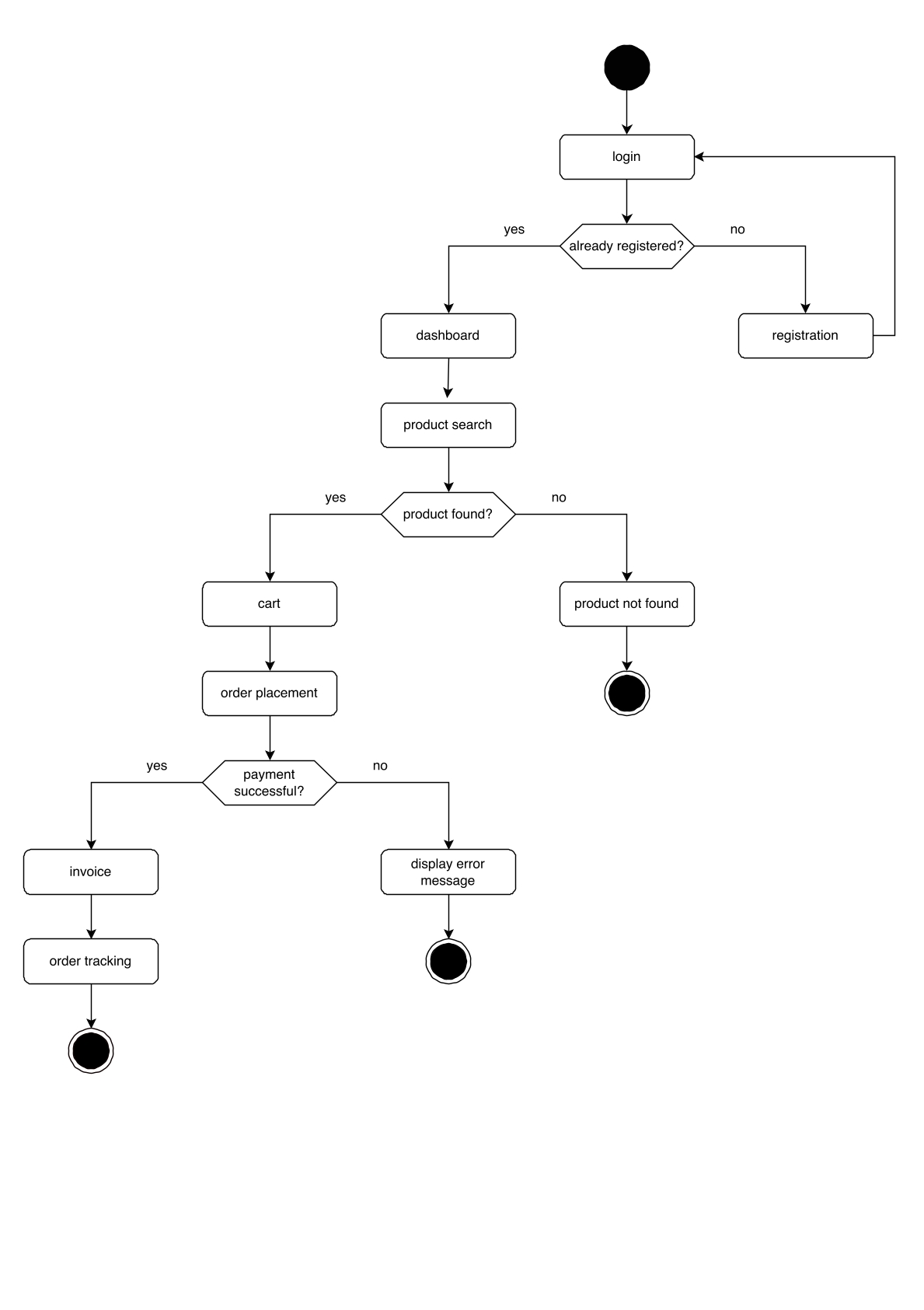
***State Chart Diagram***

## Activity Diagram

An Activity Diagram is a visual representation within UML that illustrates the flow of activities and actions in a system or process. It employs various symbols to depict tasks, decision points, concurrency, and control flows. Rectangles signify activities or tasks, while diamonds represent decision points, allowing for conditional branching. Arrows indicate the flow of control from one activity to another. Forks and joins denote concurrency, where multiple activities can occur simultaneously or in parallel. Swimlane segregate activities based on the responsible entity, facilitating clarity in complex processes. Initial and final nodes mark the commencement and completion points of the activity. Decision nodes use guards to determine the path taken based on conditions. Synchronization bars enable the coordination of parallel activities. Control flows direct the sequence of actions, while object flows depict the flow of objects between activities. Activity Diagrams serve as invaluable tools for understanding, modeling, and analyzing complex workflows in systems and processes. They offer a structured visual representation that aids in effective communication and system development.

**Key notations for Activity Diagrams:**

* **Initial Node:** Represented by a solid circle, it signifies the starting point of the activity.
* **Activity:** Shown as a rounded rectangle, it represents a task or action within the process.
* **Decision Node:** Depicted as a diamond shape, it indicates a point where the process flow can diverge based on a condition.
* **Merge Node:** Represented by a hollow diamond, it signifies a point where multiple flows converge.
* **Fork Node:** Shown as a horizontal bar, it denotes the start of concurrent activities.
* **Join Node:** Depicted as a vertical bar, it marks the point where parallel flows rejoin.
* **Final Node:** Represented by a solid circle with a border, it indicates the end of the activity.
* **Control Flow:** Arrows connecting activities, showing the sequence of actions.
* **Object Flow:** Lines with arrows representing the flow of objects between activities.
* **Swimlane:** A visual container that groups activities based on the responsible entity or system component.
* **Partition:** A horizontal or vertical area within a swimlane, further organizing activities.

****

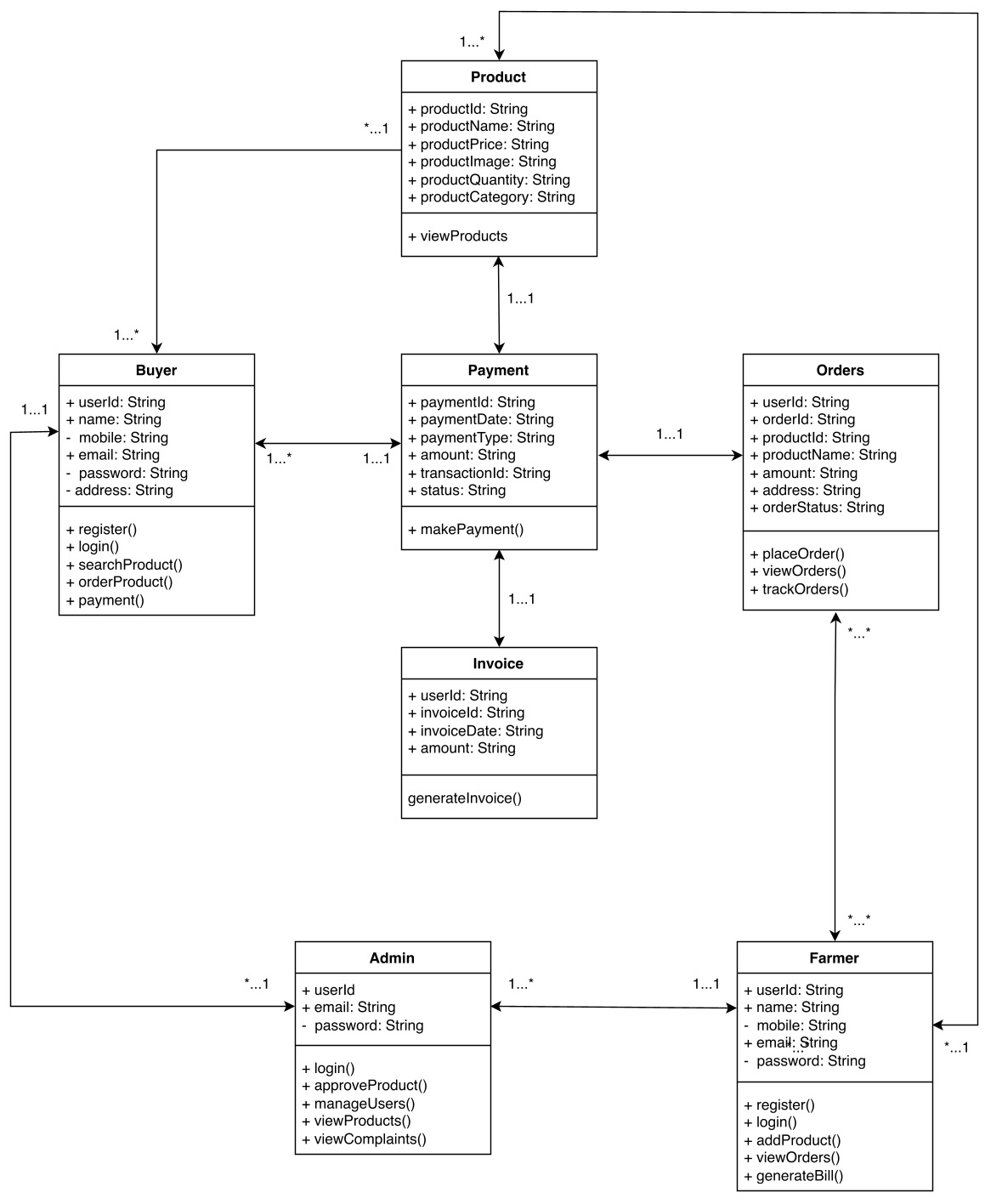
***Activity Diagram***

## Class Diagram

A Class Diagram, a fundamental tool in UML, visually represents the structure of a system by illustrating classes, their attributes, methods, and relationships. Classes, depicted as rectangles, encapsulate data and behavior within a system. Associations between classes indicate relationships, showcasing how they interact. Multiplicity notations specify the cardinality of associations. Inheritance is denoted by an arrow indicating the subclass inheriting from a super-class. Aggregation and composition illustrate whole-part relationships between classes. Interfaces, depicted as a circle, outline the contract of behavior a class must implement. Stereotypes provide additional information about a class's role or purpose. Dependencies highlight the reliance of one class on another. Association classes facilitate additional information about associations. Packages group related classes together, aiding in system organization. Class Diagrams play a pivotal role in system design, aiding in conceptualizing and planning software architectures. They serve as a blueprint for the development process, ensuring a clear and structured approach to building robust software systems.

**Key notations for Class Diagrams:**

* **Class:** Represented as a rectangle, it contains the class name, attributes, and methods.
* **Attributes:** Displayed as a list within the class, they describe the properties or characteristics of the class.
* **Methods:** Also listed within the class, they define the behaviors or operations of the class
* **Associations:** Lines connecting classes, indicating relationships and connections between them.
* **Multiplicity Notation:** Indicates the number of instances one class relates to another.
* **Inheritance:** Shown as an arrow, it signifies that one class inherits properties and behaviors from another.
* **Interfaces:** Represented by a dashed circle, they define a contract of behavior that implementing classes must follow.
* **Stereotypes:** Additional labels or annotations applied to classes to provide more information about their role or purpose.
* **Dependencies:** Shown as a dashed line with an arrow, they indicate that one class relies on another in some way.
* **Association Classes:** Represented as a class connected to an association, they provide additional information about the relationship.

****

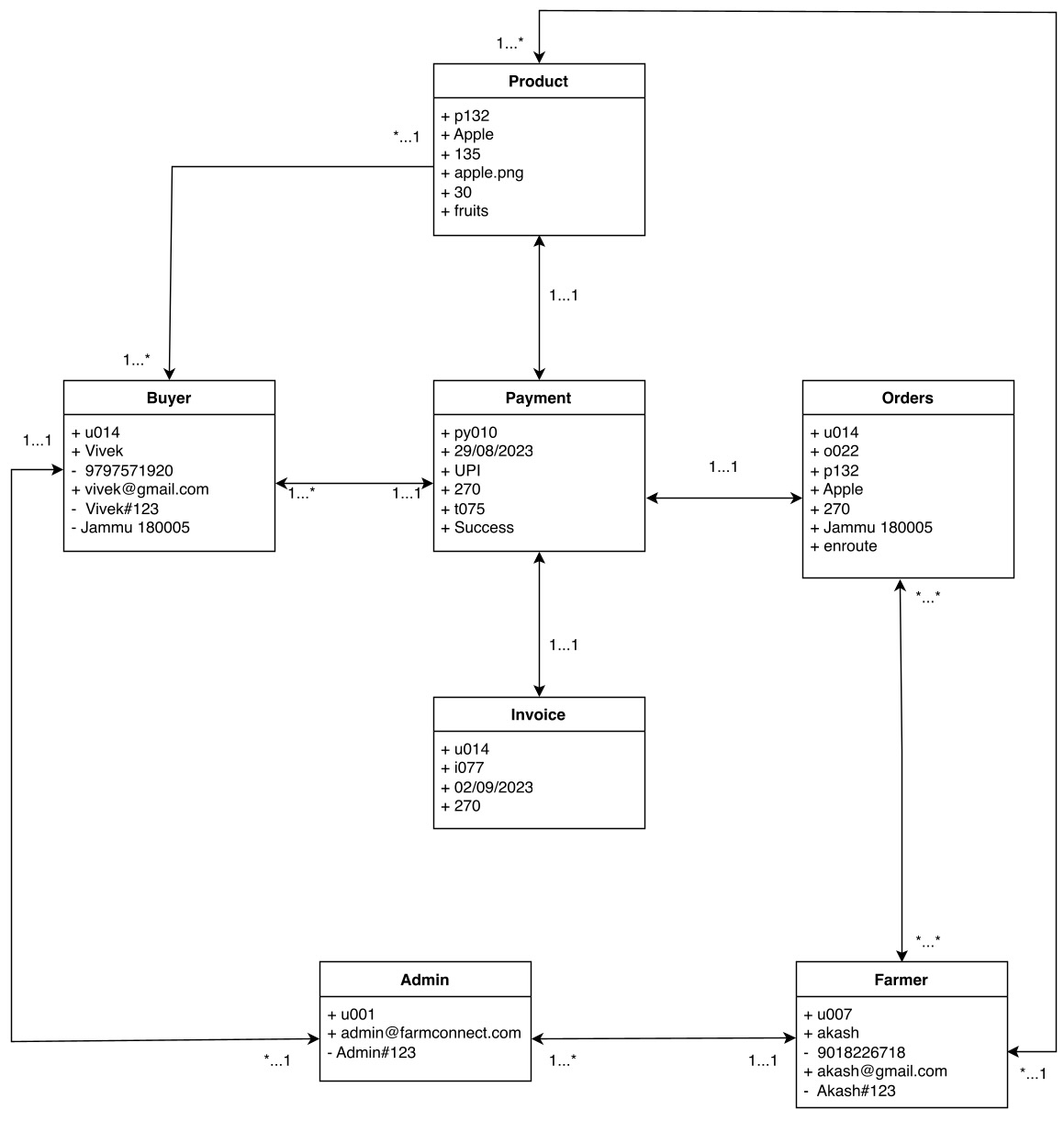
***Class Diagram***

## Object Diagram

An Object Diagram in UML provides a snapshot of a system at a specific point in time, displaying the instances of classes and their relationships. Objects, represented as rectangles, showcase the state and behavior of specific instances. Links between objects depict associations, highlighting how they interact. Multiplicity notations indicate the number of instances involved in associations. The object's state is displayed through attributes and their corresponding values. Object Diagrams offer a detailed view of runtime interactions, aiding in system understanding and testing. They focus on real-world instances, providing a tangible representation of class relationships. While similar to Class Diagrams, Object Diagrams emphasize concrete instances rather than class definitions. They serve as valuable tools for validating system design and verifying that classes and associations work as intended in practice. Object Diagrams play a crucial role in system validation, ensuring that the system's components and their interactions align with the intended design and requirements.

**Key notations for Object Diagrams:**

* **Object**: Represented as a rectangle, it contains the object's name and attributes with their values.
* **Links**: Lines connecting objects, indicating associations or relationships between them.
* **Multiplicity** **Notation**: Indicates the number of instances involved in associations.
* **Attributes with Values**: Displayed within the object, they represent the state of the object at a specific point in time.
* **Role Names**: Labels applied to associations, providing additional information about the nature of the relationship.
* **Object Name**: Represents the name of the specific instance.
* **Association End**: Indicates the end of an association, often with a role name and multiplicity.
* **Dependency Arrow**: Indicates a dependency relationship, where one object relies on another.
* **Composition Diamond**: Represents a stronger form of ownership, where one object encapsulates another.
* **Aggregation Diamond**: Signifies a whole-part relationship between objects.

****

## Component Diagram

Explanation, Diagram

**4.2.8 Deployment Diagram**

Explanation, Diagram

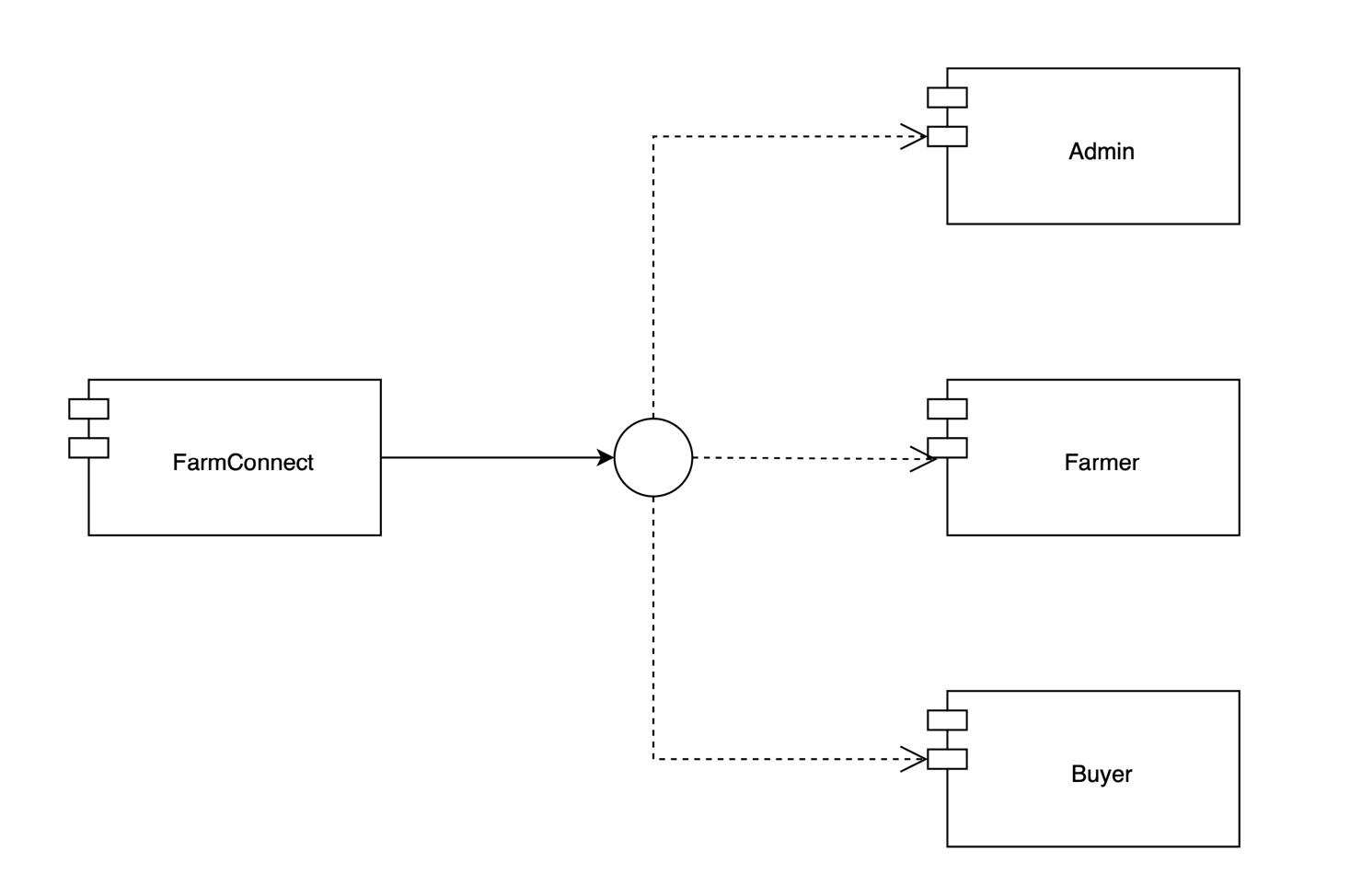
***Object Diagram***

**4.2.7 Component Diagram**

A Component Diagram, a vital aspect of UML, offers a visual representation of a system's architecture by showcasing the high-level components and their connections. Components, depicted as rectangles, encapsulate modules, classes, or even entire systems. Dependencies between components are displayed through arrows, signifying the reliance of one component on another. Interfaces, represented by a small circle, outline the services a component offers or requires. Connectors link interfaces to denote the required or provided services. Ports, depicted as small squares, serve as connection points between a component and its interfaces. Stereotypes provide additional information about the role or purpose of a component. Deployment nodes indicate the physical location or environment in which components are deployed. Component Diagrams are instrumental in system design, aiding in the organization and visualization of system architecture. They emphasize the modular structure, facilitating ease of development, maintenance, and scalability of complex software systems. Overall, Component Diagrams play a pivotal role in planning and orchestrating the architecture of sophisticated software applications.

Key notations for Component Diagrams:

* **Component**: Represented as a rectangle, it encapsulates a module, class, or system.
* **Dependency Arrow**: Indicates that one component relies on or uses another.
* **Interface**: Depicted as a small circle, it outlines the services a component offers or requires.
* **Provided and Required Interfaces**: Connectors link provided interfaces to required interfaces.
* **Port**: Shown as a small square, it serves as a connection point between a component and its interfaces.
* **Stereotypes**: Additional labels or annotations applied to components to provide more information about their role or purpose.
* **Assembly Connector**: Represents the physical connection between two components.
* **Artifact**: A physical piece of information that is used or produced by a software development process.
* **Deployment Node**: Indicates the physical location or environment in which components are deployed.
* **Manifestation Arrow**: Indicates the implementation of an interface by a component

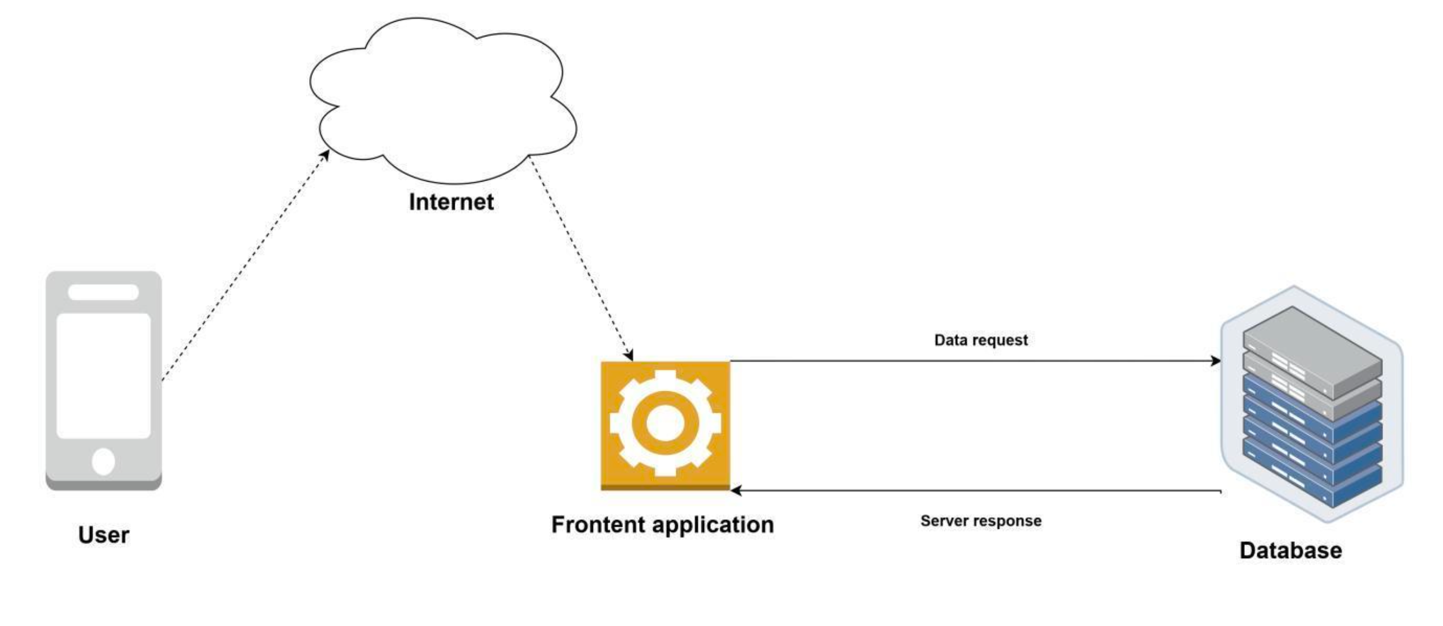


***Component Diagram***

**4.2.8 DEPLOYMENT DIAGRAM**

A Deployment Diagram, a crucial facet of UML, provides a visual representation of the physical

architecture of a system, showcasing the hardware nodes and software components. Nodes, representing hardware entities like servers or devices, are depicted as rectangles. Artifacts, denoted by rectangles with a folded corner, represent software components or files deployed on nodes. Associations between nodes and artifacts indicate the deployment of software on specific hardware. Dependencies illustrate the reliance of one node on another. Communication paths, shown as dashed lines, represent network connections between nodes. Stereotypes provide additional information about the role or purpose of nodes and artifacts. Deployment Diagrams are instrumental in system planning, aiding in the visualization and organization of hardware and software components. They emphasize the allocation of software modules to specific hardware nodes, ensuring efficient utilization of resources. Overall, Deployment Diagrams play a pivotal role in orchestrating the physical infrastructure of complex software applications.



***Deployment Diagram***

## 4.3 USER INTERFACE DESIGN USING FIGMA

|  |  |  |
| --- | --- | --- |
| figmaFarmConnect_page-0001.jpg | figmaFarmConnect_page-0002.jpg | figmaFarmConnect_page-0003.jpg |
| ***4.3.1 Splash Screen*** | ***4.3.2 Login Screen*** | ***4.3.3 SignUp Page*** |
|  |  |  |
|  |  |  |
| figmaFarmConnect_page-0005.jpg | figmaFarmConnect_page-0004.jpg | figmaFarmConnect_page-0006.jpg |
| ***4.3.4 Profile Page*** | ***4.3.5 Products Listing*** | ***9.3.6 About Page*** |

## 4.4 DATABASE DESIGN

Database Design is a critical component in the realm of information management and software development. It involves the thoughtful and systematic organization of data to ensure efficient storage, retrieval, and manipulation. A well-designed database serves as the backbone of applications, enabling them to handle large volumes of information with speed and accuracy. This process encompasses defining the structure, relationships, and constraints of data entities, optimizing for performance and scalability. Effective database design is pivotal in minimizing redundancy, ensuring data integrity, and providing a foundation for robust data analytics. It involves a deep understanding of business requirements and user needs, translating them into a coherent and logical data model. The goal of a sound database design is to create a reliable, scalable, and maintainable system that supports the organization's objectives and facilitates seamless information flow.

### 4.4.1 NoSQL Databases (NoSQL)

NoSQL databases, or "Not Only SQL," represent a transformative approach to database management. They diverge from traditional relational databases, excelling in handling large volumes of unstructured or semi-structured data. NoSQL databases are highly scalable, allowing for horizontal scaling across multiple servers, making them ideal for rapidly growing data volumes in web applications and Big Data environments. These databases come in various types, such as document-oriented, key-value stores, wide-column stores, and graph databases. Each type caters to specific use cases. Document-oriented databases, like MongoDB, are adept at storing JSON-like documents, making them popular for content management systems and real-time analytics.

One key advantage of NoSQL databases is their schema flexibility. They allow for dynamic or

semi-structured schemas, enabling data to be added or modified on the fly. This characteristic is

invaluable in projects where data structures are likely to evolve over time. NoSQL databases have found wide adoption in domains like social media, IoT applications, gaming, and real-time analytics. However, it's crucial to choose between NoSQL and traditional relational databases based on the unique requirements of the application. In essence, NoSQL databases offer a potent alternative, providing scalability, flexibility, and high performance in scenarios demanding the handling of large volumes of diverse data. Their diverse types make them indispensable tools in modern data management.

**4.4.3 Indexing**

Indexing in NoSQL databases is a crucial technique to enhance query performance and retrieval speed. Unlike relational databases, NoSQL databases utilize a variety of indexing methods tailored to different data models. In document-oriented databases like MongoDB, B-tree indexes are commonly used to accelerate search operations based on keys or fields within documents. Hash indexes, on the other hand, are prevalent in key-value stores like Redis, enabling swift retrieval of values associated with specific keys. Wide-column stores like Cassandra utilize techniques like row-level indexing to swiftly locate specific columns within a wide row. Graph databases like Neo4j employ specialized index structures optimized for graph traversal operations, allowing for rapid traversal of connected nodes.While indexing significantly improves read performance, it's important to weigh the trade-offs, as indexes can increase storage requirements and potentially slow down write operations. In summary, indexing plays a vital role in optimizing query performance and is a key aspect of designing efficient data retrieval systems in NoSQL databases.

### 4.4 TABLE/COLLECTION DESIGN

**4.4.1 Collection Name: users**

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Field Description** |
| uid | String | Unique identifier for the user |
| email | String | Email address of the user |
| name | String | Full name of the user |
| phone | String | Phone number of the user |
| ftl | String | Flag indicating whether the user is First Time Login |
| isActive | String | Indicates whether the user account is active |
| isAdminApproved | String | Indicates whether the user is approved as an admin |
| gender | String | Gender of the user |
| role | String | Role of the user |
| profileImageUrl | String | URL pointing to the user's profile image |
| street | String | Street or address details of the user |
| town | String | Town or city of the user's location |
| district | String | District of the user's location |
| state | String | State of the user's location |
| pincode | String | Pincode or postal code of the user's location |

***Collection: users***

**4.4.2 Collection Name: products**

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Field Description** |
| userId | String | Unique identifier for the user who added the product |
| productId | String | Unique identifier for the product |
| productName | String | Name of the product |
| category | String | Category of the product |
| productPrice | String | Price of the product |
| productDescription | String | Description of the product |
| productImage | String | URL pointing to the product image |
| isApproved | String | Approval status of the product, |
| stock | String | Stock quantity of the product |
| remark | String | Additional remarks or comments from Admin |

***Collection: products***

**4.4.3 Collection Name: cart**

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Field Description** |
| userId | String | Unique identifier for the user who added the product |
| productId | String | Unique identifier for the product |
| productName | String | Name of the product |
| productDescription | String | Description of the product |
| productImage | String | URL pointing to the image of the product |
| productPrice | String | Price of the product |
| quantity | String | Quantity of the product in the cart |

***Collection: cart***

**4.4.4 Collection Name: payment**

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Field Description** |
| userUid | String | Unique identifier for the user |
| paymentId | String | Unique identifier for the payment |
| customerName | String | Name of the customer |
| customerEmail | String | Email address of the customer |
| customerPhone | String | Phone number of the customer |
| amount | Number | The total amount of the transaction |
| timestamp | Timestamp | Timestamp of the transaction |
| products | Array | Array of products included in the transaction |
| products[i] | Map | Map representing the details of the first product in the array |
| products[i].productId | String | ID of the first product |
| products[i].productName | String | Name of the first product |
| products[i].quantity | Number | Quantity of the first product |
| products[i].totalPrice | Number | Total price of the first product |
| products[i].unitPrice | Number | Unit price of the first product |

***Collection: payment***

# CHAPTER 5

# SYSTEM TESTING

* 1. **INTRODUCTION**

System Testing is a crucial phase in the software development life cycle, where the entire system is evaluated against specified requirements and functionalities. It is a comprehensive and structured approach to validate that the software meets its intended objectives. This phase involves testing the integrated system as a whole to ensure that all components work together seamlessly. System Testing verifies the system's compliance with both functional and non-functional requirements, including performance, security, and usability. It is conducted in an environment that closely simulates the production environment, providing a real-world scenario for testing. The primary goal of System Testing is to identify and rectify any discrepancies or defects before the software is deployed to end-users. Through rigorous testing processes and thorough documentation, System Testing helps in delivering a reliable and high-quality software product.

Testing is the systematic process of running a program to uncover potential errors or flaws. An effective test case possesses a high likelihood of revealing previously unnoticed issues. A test is considered successful when it reveals a previously unidentified error. If a test functions as intended and aligns with its objectives, it can detect flaws in the software. The test demonstrates that the computer program is operating in accordance with its intended functionality and performing optimally. There are three primary approaches to assessing a computer program: evaluating for accuracy, assessing implementation efficiency, and analyzing computational complexity.

## TEST PLAN

A test plan is a thorough document that delineates the strategy, scope, objectives, resources, schedule, and expected outcomes for a specific testing endeavor. It functions as a guiding framework for carrying out testing activities, guaranteeing that every facet of the testing process is methodically organized and executed. Additionally, the test plan establishes the roles and responsibilities of team members, outlines the required testing environment, and sets forth the criteria for the successful completion of testing activities. This document plays a pivotal role in ensuring that the testing phase is conducted in a structured and effective manner, ultimately contributing to the overall success of the project.

**The levels of testing include:**

* Integration Testing
* Unit testing
* Validation Testing or System Testing
* Output Testing or User Acceptance Testing
* Automation Testing
* Widget Testing

### Unit Testing

Unit Testing is not only a meticulous examination of discrete units or components within a software system but also an indispensable quality assurance measure. This phase serves as a crucial foundation for the entire software testing process, where the focus lies on isolating and scrutinizing individual units of code. The objective remains unwavering: to verify that each unit performs its designated function accurately, yielding precise outputs for predefined inputs.

Moreover, Unit Testing operates independently, detached from other components, and any external dependencies are either emulated or replaced by "mock" objects, ensuring controlled evaluation. This meticulous process establishes a robust foundation for the software, confirming that each unit functions reliably and adheres meticulously to its predefined behavior.

The significance of Unit Testing cannot be overstated, as it acts as a vanguard against potential discrepancies or errors early in the development cycle. This proactive approach not only fortifies the integrity and reliability of the software but also lays the groundwork for subsequent testing phases, thereby fostering a robust and dependable software solution. This meticulous process ensures that each unit functions reliably and adheres precisely to its defined behavior. By subjecting individual code units to rigorous scrutiny, any discrepancies or errors are identified and rectified early in the development cycle, bolstering the overall integrity and reliability of the software.

### Integration Testing

Integration Testing stands as a pivotal phase in the software testing process, dedicated to scrutinizing the interactions and interfaces among diverse modules or components within a software system. Its primary objective is to ascertain that individual units of code seamlessly converge to create a unified and functional system. In stark contrast to unit testing, which assesses individual units in isolation, integration testing delves into the interplay between these units, with a keen eye for any disparities, communication glitches, or integration hurdles. By subjecting the integrated components to rigorous testing, development teams aim to affirm that these elements function cohesively, addressing any potential issues before deployment. This systematic evaluation is instrumental in ensuring that the software operates as an integrated whole, free from any unforeseen conflicts or errors that may arise from the convergence of individual modules.

### Validation Testing or System Testing

Validation Testing places the end-users at the forefront of evaluation, ensuring that the software aligns precisely with their anticipated needs and expectations. This phase stands distinct from other testing methodologies, as its primary objective is to authenticate that the software, in its final form, serves its intended purpose seamlessly within the real-world scenarios it was designed for. As a culmination of the testing process, Validation Testing carries the responsibility of confirming that the software not only meets the defined technical specifications but also delivers genuine value to its users. It does so by scrutinizing the software against the backdrop of actual usage, thereby fortifying its readiness for deployment. Moreover, in Validation Testing, user stories and acceptance criteria form the cornerstone of assessment. Stakeholders' expectations are meticulously validated, ensuring that every specified requirement is met. Additionally, beta testing, a common practice in this phase, involves a select group of end-users testing the software in a live environment, providing invaluable feedback that can inform potential refinements.

### Output Testing or User Acceptance Testing.

Output Testing, also known as Results Validation, is a critical phase in the software testing process. Its primary focus is to verify the correctness and accuracy of the output generated by a software application. The goal is to ensure that the system produces the expected results for a given set of inputs and conditions.

**Key aspects of Output Testing include:**

* **Comparison with Expected Results**: This phase involves comparing the actual output of the software with the expected or predefined results.
* **Test Case Design:** Test cases are designed to cover various scenarios and conditions to thoroughly evaluate the accuracy of the output.
* **Validation Criteria**: The criteria for validating the output are typically defined during the requirements and design phase of the software development process.
* **Regression Testing**: Output Testing often includes regression testing to ensure that changes or updates to the software do not affect the correctness of the output.
* **Data Integrity**: It verifies that data is processed and displayed correctly, without any corruption or loss.
* **Precision and Completeness**: Output Testing assesses not only the precision of the results but also their completeness in addressing the requirements.
* **Error Handling**: It evaluates how the system handles errors or exceptions and ensures that appropriate error messages are displayed.
  + 1. **Automation Testing**

Automation Testing stands as a cornerstone in the software testing process, harnessing the power of automated tools and scripts to meticulously execute test cases. In stark contrast to manual testing, which hinges on human intervention, automation testing brings forth a streamlined approach, employing software to conduct repetitive, intricate, and time-consuming tests. This methodology not only heightens operational efficiency but also significantly diminishes the likelihood of human error, ensuring precise and reliable results. Moreover, it empowers thorough testing across a diverse array of scenarios and configurations, from browser compatibility to load and performance assessments.

By automating the testing process, organizations can realize a myriad of benefits. It enables the seamless execution of regression tests, providing confidence that existing functionalities remain intact after each round of enhancements or modifications. Furthermore, automation facilitates the concurrent execution of multiple tests, thereby expediting the overall testing cycle. This approach is particularly invaluable in environments characterized by rapid development and frequent software updates, such as Agile and DevOps setups.

* + 1. **Widget Testing**

Widget Testing in Flutter is a critical step towards building robust and reliable user interfaces. By isolating and scrutinizing individual widgets, developers gain confidence in the functionality and appearance of each component. This level of granularity allows for precise testing, ensuring that widgets respond correctly to various user interactions and scenarios.

One of the key advantages of Widget Testing is the ability to mock dependencies. This means that external services or resources that the widget relies on can be simulated, allowing for controlled and predictable testing environments. Additionally, developers can set expectations and employ assertions to validate the widget's behavior under different conditions. Furthermore, Widget Testing is seamlessly integrated with popular testing frameworks like `flutter\_test`. This ensures that tests can be organized efficiently and run as part of the automated testing process, providing rapid feedback on the status of widgets.

Another important facet of Widget Testing is the inclusion of Golden Tests. These tests capture screenshots of the widget's visual appearance and compare them against reference images. This helps maintain visual consistency, making sure that UI elements render consistently across different devices and screen sizes.

**Test Case 1**

**Code**

**Screenshot Test Case 1**

**Test Report**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Case 1** | | | | | |
| **Project Name:** | | | | | |
| **Login Test Case** | | | | | |
| **Test Case ID: Test\_1** | | | **Test Designed By:** | | |
| **Test Priority(Low/Medium/High):** | | | **Test Designed Date:** | | |
| **Module Name**: | | | **Test Executed By :** | | |
| **Test Title :** | | | **Test Execution Date:** | | |
| **Description:** | | |  | | |
| **Pre-Condition :**User has valid username and password | | | | | |
| **Step** | **Test Step** | **Test Data** | **Expected Result** | **Actual Result** | **Status(Pass/**  **Fai l)** |
| 1 |  |  |  |  |  |
| 2 |  |  |  |  |  |
| 3 |  |  |
| 4 |  |  |
| 5 |  |  |  |  |  |
|  |  |  |  |  |  |
| 6 |  |  |
| 7 |  |  |  |  |  |
|  |  |  |  |  |
| **Post-Condition:** | | | | | |

# CHAPTER 6

# IMPLEMENTATION

## INTRODUCTION

Project implementation is the phase where plans and strategies transform into tangible actions and outcomes. It marks the transition from theoretical concepts to practical application. This pivotal stage requires meticulous planning, resource allocation, and a dedicated team to execute tasks according to the established timeline and objectives. In this phase, the project team translates the project's blueprints into real-world activities, ensuring that each step aligns with the overarching goals. Effective project implementation demands clear communication, robust leadership, and a keen eye for detail. This introduction sets the stage for a comprehensive understanding of the project implementation process, emphasizing its significance in achieving the envisioned goals. As we delve deeper, we will explore key components, strategies, and best practices that contribute to successful project implementation.

The crux of successful project implementation lies not just in technical proficiency, but also in the art of effective communication. Clear channels of dialogue serve as the lifeblood that sustains the project's momentum, fostering synergy among team members and stakeholders alike. A bedrock of robust leadership provides the necessary guidance and inspiration, steering the ship through uncharted waters with confidence and purpose. Additionally, an unyielding commitment to detail acts as the linchpin that secures the integrity of each executed task.

This introduction sets the stage for a profound comprehension of the project implementation process, underlining its indomitable significance in realizing the envisioned goals. As we embark on this journey of exploration, we will unfurl the tapestry of key components, unveil strategies, and illuminate best practices that form the crucible of triumphant project implementation.

**The implementation state involves the following tasks:**

* Careful planning.
* Investigation of system and constraints.
* Design of methods to achieve the changeover.

## IMPLEMENTATION PROCEDURES

### User Training

User training is a critical component of ensuring the effective utilization of any application software. It involves imparting the necessary knowledge and skills to end-users, enabling them to navigate and utilize the software efficiently. This training equips users with a comprehensive understanding of the software's features, functions, and capabilities. Through hands-on sessions and guided tutorials, users learn how to perform tasks, customize settings, and troubleshoot common issues. Moreover, user training fosters confidence and proficiency, empowering individuals to maximize their productivity while using the application. Regular updates and refresher sessions further enhance user competence, ensuring they stay abreast of new features and functionalities.

### Training on the Application Software

Training on the application software is a structured program designed to familiarize individuals with the intricacies and functionalities of a specific software application. It encompasses a range of topics, from basic navigation to advanced features, tailored to meet the diverse needs of users. This training often includes interactive demonstrations, hands-on exercises, and Q&A sessions to facilitate effective learning. Trainers may also provide supplemental resources such as user manuals or online guides for reference. By the end of the training, participants are equipped skills and knowledge required to proficiently utilize the application software in their respective contexts.

### System Maintenance

### System maintenance is a crucial aspect of ensuring the seamless operation and longevity of any software application. It encompasses a series of tasks aimed at monitoring, optimizing, and troubleshooting the underlying infrastructure on which the application runs. This includes activities such as regular software updates, performance monitoring, and data backups. Additionally, system maintenance involves identifying and rectifying any potential vulnerabilities or inefficiencies that may impede the software's performance. Proactive maintenance measures contribute to a stable and secure environment, minimizing the risk of unexpected downtime or data loss.

### Installation

### Installing this application is a straightforward process that grants accessibility on mobile devices supporting API versions from 29 to 30. Here's a concise guide:

### Compatibility Verification: Ensure your mobile device runs on API version 29 to 30 for seamless installation and operation.

### Download the APK: Obtain the APK file from a trusted source, ensuring it matches the application's official version.

### Adjust Security Settings: If needed, enable 'Unknown Sources' in your device's settings to allow APK installations.

### Locate and Run the APK: Access the downloaded APK file, typically found in the 'Downloads' folder, and tap to initiate installation.

### Permissions and Installation: Grant necessary permissions for the application to function optimally and proceed with the installation process.

### Quick Installation: APKs often install swiftly, bypassing the need for extensive downloads from official app stores.

### Independence from App Stores: APKs offer the advantage of being downloadable from various sources, granting users flexibility in obtaining apps.

### Beta and Unreleased Versions: Users can access beta or unreleased versions of applications via APKs, providing a preview of upcoming features.

### Offline Installation: APKs can be shared and installed without an active internet connection, enhancing accessibility.

Screenshots

---------------------

# CHAPTER 7

# CONCLUSION AND FUTURE SCOPE

## CONCLUSION

FarmConnect emerges as a pivotal solution in the agricultural landscape, introducing a digital platform that seamlessly connects farmers and buyers. With a robust technology stack comprising Flutter for the front-end and Firebase for the back-end, FarmConnect transcends traditional supply chain models, providing a user-friendly interface for farmers to showcase their products and buyers to access the freshest farm produce directly.

The key modules, ranging from the comprehensive Admin functionalities to the specific tools tailored for Farmers and Buyers, underscore the platform's commitment to transparency, efficiency, and sustainability. Admin functionalities empower platform administrators to ensure a smooth and secure user experience, from managing user accounts to facilitating secure transactions. Farmers benefit from tools for efficient inventory management, order fulfillment, and personalized guidance for sustainable farming. Buyers, on the other hand, enjoy a user-friendly interface for seamless product discovery, order placement, and real-time order tracking.

FarmConnect's functionalities extend beyond mere commerce. The inclusion of advanced features such as image recognition for search, disease detection, pest control guidance, and fertilizer recommendations showcases a commitment to leveraging technology for the holistic improvement of farming practices. The platform not only facilitates transactions but also fosters a community where farmers and buyers can engage directly, inquire about products, and provide feedback.

* 1. **FUTURE SCOPE**

As FarmConnect progresses into future stages of development, the integration of additional functionalities promises to enhance the platform's capabilities even further. In the upcoming stages, it is envisioned that:

* **Integration of Artificial Intelligence (AI) and Machine Learning (ML):** Implementing AI and ML algorithms can enhance user experience by providing personalized service recommendations based on user preferences and behavior patterns
* **Expansion of Categories:** Introducing additional categories based on user feedback and evolving market trendscan further diversify the offerings and meet a wider range of user needs.
* **Chatbot Integration:** Enabling voice assistants or chatbots can streamline user interactions.
* **Filtering Options:** Further refinement of filtering options, allowing buyers to tailor searches based on specific criteria such as product attributes, farm practices, and certifications.
* **Communications and Support Enhancements**: Introduction of features for efficient communication and support, including ticketing systems, real-time chat support, and comprehensive FAQs.
* **AI-Powered Disease Detection**: Advancement of disease detection algorithms using artificial intelligence, enabling more accurate and early identification of crop diseases.

# CHAPTER 8

# BIBLIOGRAPHY

### REFERENCES:

* Gary B. Shelly, Harry J. Rosenblatt, “System Analysis and Design”, 2009.
* Roger S Pressman, “Software Engineering”, 1994.
* PankajJalote, “Software engineering: a precise approach”, 2006.
* James lee and Brent ware Addison, “Open source web development with LAMP”, 2003
* IEEE Std 1016 Recommended Practice for Software Design Descriptions.

### WEBSITES:

* https://pub.dev/
* https://console.firebase.google.com
* https://docs.flutter.dev/development/ui/widgets
* https://fluttergems.dev/
* https://chat.openai.com/
* https://www.github.com/
* https://www.youtube.com/
* https://draw.io/
* https://medium.com/

# CHAPTER 9

# APPENDIX

## Sample Code

**Application Entry Point - main.dart**

import 'package:farmconnect/features/…/AdminPages/adminDashboard.dart';

import 'package:farmconnect/features/…/AdminPages/displayBuyers.dart';

import 'package:farmconnect/features/…/AdminPages/displayFarmersApproved.dart';

import 'package:farmconnect/features/…/AdminPages/displayFarmersRejected.dart';

import 'package:farmconnect/features/…/AdminPages/displayFramersPending.dart';

import 'package:farmconnect/features/…/AdminPages/displayProductsCaterogyWise.dart';

import 'package:farmconnect/features/…/AdminPages/paymentSuccessful.dart';

import 'package:farmconnect/features/…/AdminPages/productsApproved.dart';

import 'package:farmconnect/features/…/AdminPages/productsPending.dart';

import 'package:farmconnect/features/…/AdminPages/productsRejected.dart';

import 'package:farmconnect/features/…/BuyerPages/buyerDashboard.dart';

import 'package:farmconnect/features/…/BuyerPages/buyerProfile.dart';

import 'package:farmconnect/features/…/BuyerPages/buyerftl.dart';

import 'package:farmconnect/features/…/BuyerPages/productsDairy.dart';

import 'package:farmconnect/features/…/BuyerPages/productsFruit.dart';

import 'package:farmconnect/features/…/BuyerPages/productsPoultry.dart';

import 'package:farmconnect/features/…/BuyerPages/productsVegetable.dart';

import 'package:farmconnect/features/…/BuyerPages/updatePassword.dart';

import 'package:farmconnect/features/…/BuyerPages/updateProfile.dart';

import 'package:farmconnect/features/…/Cart/myOrders.dart';

import 'package:farmconnect/features/…/Cart/viewBillsandInvoice.dart';

import 'package:farmconnect/features/…/Cart/cartProvider.dart';

import 'package:farmconnect/features/…/FarmerPages/farmerDashboard.dart';

import 'package:farmconnect/features/…/FarmerPages/farmerftl.dart';

import 'package:farmconnect/features/…/admin\_approval\_pending.dart';

import 'package:farmconnect/features/…/common/termsAndConditions.dart';

import 'package:firebase\_auth/firebase\_auth.dart';

import 'package:firebase\_core/firebase\_core.dart';

import 'package:flutter/material.dart';

import 'package:farmconnect/features/app/splash\_screen/splash\_screen.dart';

import 'package:farmconnect/features/…/login\_page.dart';

import 'package:farmconnect/features/…/sign\_up\_page.dart';

import 'package:farmconnect/features/…/email\_verification\_pending\_page.dart';

import 'package:provider/provider.dart';

Future<void> main() async {

WidgetsFlutterBinding.ensureInitialized();

await Firebase.initializeApp();

final cartProvider = CartProvider();

final user = FirebaseAuth.instance.currentUser;

if (user != null) {

cartProvider.setUserId(user.uid);

await cartProvider.initializeCartFromFirestore();

}

runApp(

ChangeNotifierProvider.value(

value: cartProvider,

child: MyApp(),

),

);

}

class MyApp extends StatelessWidget {

@override

Widget build(BuildContext context) {

final cartProvider = Provider.of<CartProvider>(context);

final user = FirebaseAuth.instance.currentUser;

if (user != null) {

cartProvider.setUserId(user.uid);

}

return MaterialApp(

debugShowCheckedModeBanner: false,

title: 'FarmConnect',

initialRoute: '/',

routes: {

'/': (context) => SplashScreen(

child: LoginPage(),

),

'/login': (context) => LoginPage(),

'/signUp': (context) => SignUpPage(),

'/buyer\_profile': (context) => BuyerProfilePage(),

'/buyer\_home' : (context) => BuyerDashboard(),

'/update\_details': (context) => UpdateDetailsPage(),

'/update\_password': (context) => UpdatePasswordPage(),

'/farmer\_dash': (context) => FarmerDashboard(),

'/buyer\_ftl' : (context) => BuyerFTLPage(),

'/farmer\_ftl' : (context) => FarmerFTLPage(),

'/admin\_dashboard' : (context) => AdminDashboard(),

'/buyer\_details' : (context) => BuyerDetailsPage(),

'/farmer\_details' : (context) => FarmerDetailsPage(),

'/email\_verification\_pending': (context) => EmailVerificationPendingPage(),

'/terms' : (context) => TermsPage(),

'/pendingapproval' : (context) => PendingApprovalPage(),

'/approvedproducts' : (context) => ApprovedProductsPage(),

'/rejectedproducts' : (context) => RejectedProductsPage(),

'/poultry\_page' : (context) => PoultryProductsPage(),

'/dairy\_page' : (context) => DairyProductsPage(),

'/fruits\_page' : (context) => FruitsProductsPage(),

'/vegetables\_page' : (context) => VegetableProductsPage(),

'/added\_product' : (context) => FarmerDashboard(),

'/admin\_approval\_pending' : (context) => AdminApprovalPendingPage(),

'/farmer\_approval\_pending' : (context) => PendingFarmerApprovalPage(),

'/farmer\_approval\_rejected' : (context) => RejectedFarmerApprovalPage(),

'/products\_categoryWise' : (context) => CategoryWiseProducts(),

"/bills\_and\_invoice" : (context) => BillsPage(),

"/my\_orders" : (context) => MyOrdersPage(),

'/paymentSuccessful' : (context) => PaymentSuccessfulPage(),

},

);

}

}

**Admin Dashboard – adminDashboard.dart**

import 'package:firebase\_auth/firebase\_auth.dart';

import 'package:flutter/material.dart';

import 'package:cloud\_firestore/cloud\_firestore.dart';

import 'package:google\_sign\_in/google\_sign\_in.dart';

class AdminDashboard extends StatefulWidget {

@override

\_AdminDashboardState createState() => \_AdminDashboardState();

}

class \_AdminDashboardState extends State<AdminDashboard> {

int numberOfBuyers = 0;

int numberOfFarmers = 0;

int farmerApprovalPending = 0;

int farmerApprovalRejected = 0;

int pendingProducts = 0;

int approvedProducts = 0;

int rejectedProducts = 0;

int numberOfPayments = 0;

@override

void initState() {

super.initState();

fetchUserCounts();

fetchProductCounts();

fetchPaymentCounts();

}

Future<void> fetchPaymentCounts() async {

final paymentsCollection = FirebaseFirestore.instance.collection('payments');

final paymentQuery = await paymentsCollection.get();

numberOfPayments = paymentQuery.docs.length;

setState(() {});

}

Future<void> fetchUserCounts() async {

final usersCollection = FirebaseFirestore.instance.collection('users');

final buyerQuery = await usersCollection.where('role', isEqualTo: 'Buyer').get();

numberOfBuyers = buyerQuery.docs.length;

final farmerApprovedQuery = await usersCollection

.where('role', isEqualTo: 'Farmer')

.where('isAdminApproved', isEqualTo: 'approved')

.get();

numberOfFarmers = farmerApprovedQuery.docs.length;

final farmerPendingQuery = await usersCollection

.where('role', isEqualTo: 'Farmer')

.where('isAdminApproved', isEqualTo: 'pending')

.get();

farmerApprovalPending = farmerPendingQuery.docs.length;

final farmerRejectedQuery = await usersCollection

.where('role', isEqualTo: 'Farmer')

.where('isAdminApproved', isEqualTo: 'rejected')

.get();

farmerApprovalRejected = farmerRejectedQuery.docs.length;

setState(() {});

}

Future<void> fetchProductCounts() async {

final productsCollection = FirebaseFirestore.instance.collection('products');

final pendingQuery = await productsCollection.where('isApproved', isEqualTo: 'Pending').get();

pendingProducts = pendingQuery.docs.length;

final approvedQuery = await productsCollection.where('isApproved', isEqualTo: 'Approved').get();

approvedProducts = approvedQuery.docs.length;

final rejectedQuery = await productsCollection.where('isApproved', isEqualTo: 'Rejected').get();

rejectedProducts = rejectedQuery.docs.length;

setState(() {});

}

Future<void> \_refreshData() async {

await fetchUserCounts();

await fetchProductCounts();

}

@override

Widget build(BuildContext context) {

return Scaffold(

backgroundColor: Colors.black,

appBar: AppBar(

backgroundColor: Colors.blueGrey[900],

title: Text(

"Admin Dashboard",

style: TextStyle(

color: Colors.green,

fontSize: 20.0,

fontWeight: FontWeight.bold,

),

),

automaticallyImplyLeading: false,

actions: [

IconButton(

icon: Icon(Icons.logout),

color: Colors.red,

onPressed: () async {

final FirebaseAuth \_auth = FirebaseAuth.instance;

final GoogleSignIn googleSignIn = GoogleSignIn();

try {

await \_auth.signOut();

await googleSignIn.signOut();

Navigator.pushNamed(context, "/login");

} catch (error) {

print("Error signing out: $error");

}

},

),

IconButton(

icon: Icon(Icons.refresh),

color: Colors.green,

onPressed: () {

\_refreshData();

},

),

],

),

body: Container(

child: Stack(

children: [

GridView.count(

crossAxisCount: 2,

padding: EdgeInsets.all(16.0),

children: [

InkWell(

child: AdminDashboardTile(

title: "Farmers",

count: numberOfFarmers,

tileColor: Colors.blue,

iconData: Icons.people,

gradientColors: [Colors.blue.shade300, Colors.blue.shade900],

),

onTap: () {

Navigator.pushNamed(context, '/farmer\_details');

},

),

InkWell(

onTap: () {

Navigator.pushNamed(context, '/buyer\_details');

},

child: AdminDashboardTile(

title: "Buyers",

count: numberOfBuyers,

tileColor: Colors.blue,

iconData: Icons.shopping\_cart,

gradientColors: [Colors.blue.shade300, Colors.blue.shade900],

),

),

InkWell(

onTap: () {

Navigator.pushNamed(context, '/farmer\_approval\_pending');

},

child: AdminDashboardTile(

title: "Farmer Approval\n Pending",

count: farmerApprovalPending,

tileColor: Colors.orange,

iconData: Icons.pending,

gradientColors: [Colors.orange.shade300, Colors.orange.shade900],

),

),

InkWell(

onTap: () {

Navigator.pushNamed(context, '/farmer\_approval\_rejected');

},

child: AdminDashboardTile(

title: "Farmer Approval\n Rejected",

count: farmerApprovalRejected,

tileColor: Colors.red,

iconData: Icons.cancel,

gradientColors: [Colors.red.shade300, Colors.red.shade900],

),

),

InkWell(

onTap: () {

Navigator.pushNamed(context, '/pendingapproval');

},

child: AdminDashboardTile(

title: " Pending\nApproval",

count: pendingProducts,

tileColor: Colors.orange,

iconData: Icons.timer,

gradientColors: [Colors.orange.shade300, Colors.orange.shade900],

),

),

InkWell(

onTap: () {

Navigator.pushNamed(context, '/approvedproducts');

},

child: AdminDashboardTile(

title: "Approved\nProducts",

count: approvedProducts,

tileColor: Colors.green,

iconData: Icons.check\_circle,

gradientColors: [Colors.green.shade300, Colors.green.shade900],

),

),

InkWell(

onTap: () {

Navigator.pushNamed(context, '/rejectedproducts');

},

child: AdminDashboardTile(

title: "Rejected\nProducts",

count: rejectedProducts,

tileColor: Colors.red,

iconData: Icons.cancel,

gradientColors: [Colors.red.shade300, Colors.red.shade900],

),

),

InkWell(

onTap: () {

Navigator.pushNamed(context, '/products\_categoryWise');

},

child: AdminDashboardTile(

title: " Products\nCategory Wise",

count: 4,

tileColor: Colors.red,

iconData: Icons.shopping\_cart,

gradientColors: [Colors.blue, Colors.blue],

),

),

InkWell(

onTap: () {

Navigator.pushNamed(context, '/paymentSuccessful');

},

child: AdminDashboardTile(

title: " Payments\nSuccessful",

count: numberOfPayments,

tileColor: Colors.red,

iconData: Icons.payments,

gradientColors: [Colors.blue, Colors.blue],

),

),

InkWell(

onTap: () {

Navigator.pushNamed(context, '/testing');

},

child: AdminDashboardTile(

title: "Testing",

count: 0,

tileColor: Colors.grey,

iconData: Icons.sms\_failed\_rounded,

gradientColors: [Colors.grey, Colors.grey],

),

),

],

),

],

),

),

);

}

}

class AdminDashboardTile extends StatelessWidget {

final String title;

final int count;

final Color tileColor;

final IconData iconData;

final List<Color> gradientColors;

const AdminDashboardTile({

required this.title,

required this.count,

required this.tileColor,

required this.iconData,

required this.gradientColors,

});

@override

Widget build(BuildContext context) {

return Card(

shape: RoundedRectangleBorder(

borderRadius: BorderRadius.circular(25.0),

),

child: Container(

decoration: BoxDecoration(

gradient: LinearGradient(

begin: Alignment.topLeft,

end: Alignment.bottomRight,

colors: gradientColors,

),

),

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: [

Icon(

iconData,

color: Colors.white,

size: 40.0,

),

SizedBox(height: 10.0),

Text(

title,

style: TextStyle(

color: Colors.white,

fontSize: 24.0,

fontWeight: FontWeight.bold,

),

),

SizedBox(height: 10.0),

Center(

child: Text(

count.toString(),

style: TextStyle(

color: Colors.white,

fontSize: 36.0,

fontWeight: FontWeight.bold,

),

),

),

],

),

),

);

}

}

**Farmer Dashboard – farmerDashboard.dart**

import 'package:farmconnect/features/…/FarmerPages/addproducts.dart';

import 'package:farmconnect/features/…/FarmerPages/farmer\_page.dart';

import 'package:farmconnect/features/…/FarmerPages/myproducts.dart';

import 'package:flutter/material.dart';

import 'package:firebase\_auth/firebase\_auth.dart';

import 'package:google\_sign\_in/google\_sign\_in.dart';

class FarmerDashboard extends StatefulWidget {

const FarmerDashboard({Key? key}) : super(key: key);

@override

\_FarmerDashboardState createState() => \_FarmerDashboardState();

}

class \_FarmerDashboardState extends State<FarmerDashboard> {

@override

Widget build(BuildContext context) {

return DefaultTabController(

length: 3,

child: Scaffold(

appBar: AppBar(

automaticallyImplyLeading: false,

title: Text(

"Farmer Dashboard",

style: TextStyle(

color: Colors.green,

fontSize: 20,

fontWeight: FontWeight.bold,

),

),

backgroundColor: Colors.blueGrey[900],

actions: <Widget>[

IconButton(

icon: Icon(Icons.logout),

color: Colors.red,

onPressed: () async {

final FirebaseAuth \_auth = FirebaseAuth.instance;

final GoogleSignIn googleSignIn = GoogleSignIn();

try {

await \_auth.signOut();

await googleSignIn.signOut();

Navigator.pushNamed(context, "/login");

} catch (error) {

print("Error signing out: $error");

}

},

),

],

bottom: TabBar(

tabs: [

Tab(

text: "Add Products",

icon: Icon(Icons.add),

),

Tab(

text: "My Products",

icon: Icon(Icons.store),

),

Tab(

text: "Profile",

icon: Icon(Icons.person),

),

],

indicatorColor: Colors.green,

),

),

backgroundColor: Colors.black,

body: TabBarView(

children: <Widget>[

AddProducts(),

MyProductsPage(),

FarmerPage(),

],

),

),

);

}

}

**Buyer Dashboard – buyerDashboard.dart**

import 'package:farmconnect/features/…/BuyerPages/buyerProfile.dart';

import 'package:farmconnect/features/…/BuyerPages/productSearch.dart';

import 'package:farmconnect/features/…/Cart/cartPage.dart';

import 'package:flutter/material.dart';

import 'package:firebase\_auth/firebase\_auth.dart';

import 'package:google\_sign\_in/google\_sign\_in.dart';

import 'package:cloud\_firestore/cloud\_firestore.dart';

import 'productsDairy.dart';

import 'productsFruit.dart';

import 'productsPoultry.dart';

import 'productsVegetable.dart';

final TextEditingController \_searchController = TextEditingController();

class BuyerDashboard extends StatefulWidget {

const BuyerDashboard({Key? key}) : super(key: key);

@override

\_BuyerDashboardState createState() => \_BuyerDashboardState();

}

class \_BuyerDashboardState extends State<BuyerDashboard> {

late Future<List<String>> productList;

Future<List<String>> getProductList() async {

List<String> productList = [];

try {

QuerySnapshot querySnapshot =

await FirebaseFirestore.instance.collection('products').get();

for (QueryDocumentSnapshot documentSnapshot in querySnapshot.docs) {

String productName = documentSnapshot['productName'];

productList.add(productName);

}

} catch (error) {

print("Error fetching product list: $error");

}

return productList;

}

Future<void> \_signOut(BuildContext context) async {

final FirebaseAuth \_auth = FirebaseAuth.instance;

final GoogleSignIn googleSignIn = GoogleSignIn();

try {

await \_auth.signOut();

await googleSignIn.signOut();

Navigator.popAndPushNamed(context, "/login");

} catch (error) {

print("Error signing out: $error");

}

}

@override

void initState() {

super.initState();

productList = getProductList();

}

@override

Widget build(BuildContext context) {

return DefaultTabController(

length: 5,

child: Scaffold(

floatingActionButton: FloatingActionButton(

onPressed: () {

Navigator.push(

context, MaterialPageRoute(builder: (context) => CartPage()));

},

child: Icon(Icons.shopping\_cart),

),

backgroundColor: Colors.black,

appBar: AppBar(

automaticallyImplyLeading: false,

title: Text(

"Buyer Dashboard",

style: TextStyle(

color: Colors.green, fontSize: 20,

fontWeight: FontWeight.bold,

),

),

backgroundColor: Colors.lightBlue[900],

actions: <Widget>[

Container(

margin: EdgeInsets.only(right: 16.0), // Adjust margin as needed

child: InkWell(

onTap: () {

showSearch(

context: context,

delegate: ProductSearch(productList),

);

},

child: Padding(

padding: const EdgeInsets.all(8.0),

child: Icon(

Icons.search,

size: 30.0,

color: Colors.white,

),

),

),

),

ElevatedButton(

onPressed: () => \_signOut(context),

style: ButtonStyle(

backgroundColor: MaterialStateProperty.all(Colors.red),

),

child: Row(

children: [

Icon(

Icons.logout,

color: Colors.white,

),

SizedBox(width: 8.0),

Text(

'Sign Out',

style: TextStyle(

fontSize: 16,

fontWeight: FontWeight.bold,

color: Colors.white,

),

),

],

),

),

],

bottom: TabBar(

isScrollable: true,

tabs: [

Tab(

icon: Icon(Icons.shopping\_basket),

text: 'Dairy',

),

Tab(

icon: Icon(Icons.shopping\_basket),

text: 'Poultry',

),

Tab(

icon: Icon(Icons.shopping\_basket),

text: 'Fruits',

),

Tab(

icon: Icon(Icons.shopping\_basket),

text: 'Vegetables',

),

Tab(

icon: Icon(Icons.person),

text: 'Profile',

),

],

),

),

body: FutureBuilder<List<String>>(

future: productList,

builder: (context, snapshot) {

if (snapshot.connectionState == ConnectionState.waiting) {

return Center(child: CircularProgressIndicator());

} else if (snapshot.hasError) {

return Center(child: Text('Error: ${snapshot.error}'));

} else {

return TabBarView(

children: [

DairyProductsPage(),

PoultryProductsPage(),

FruitsProductsPage(),

VegetableProductsPage(),

BuyerProfilePage(),

],

);

}

},

),

),

);

}

}

**Cart Provider – cartProvider.dart**

import 'package:flutter/material.dart';

import 'package:cloud\_firestore/cloud\_firestore.dart';

class CartProvider extends ChangeNotifier {

String? \_userId;

List<Map<String, dynamic>> \_cartItems = [];

List<Map<String, dynamic>> get cartItems => \_cartItems;

void setUserId(String userId) {

\_userId = userId;

}

Future<void> fetchCartFromFirestore() async {

try {

final userCartCollection = FirebaseFirestore.instance.collection('cart').doc(\_userId);

final cartData = await userCartCollection.get();

if (cartData.exists) {

\_cartItems = List.from(cartData['cartItems']);

notifyListeners();

}

} catch (error) {

print('Error fetching cart from Firestore: $error');

}

}

Future<void> addToCart(Map<String, dynamic> item) async {

final userCartCollection = FirebaseFirestore.instance.collection('cart').doc(\_userId);

item['quantity'] = item['quantity'] ?? 1;

if (!\_cartItems.contains(item)) {

final productCollection = FirebaseFirestore.instance.collection('products');

final productData = await productCollection.doc(item['productId']).get();

int productStock = productData['stock'] ?? 0;

if (productStock >= item['quantity']) {

await \_updateStock(item['productId'], item['quantity'], false);

\_cartItems.add(item);

notifyListeners();

await userCartCollection.set({'cartItems': \_cartItems});

} else {

print('Insufficient stock for ${item['productName']}');

}

} else {

print('${item['productName']} is already in the cart');

}

}

double? totalAmount() {

double total = 0.0;

for (var item in \_cartItems) {

double productPrice = item['productPrice']?.toDouble() ?? 0.0;

total += productPrice \* (item['quantity'] ?? 1);

}

return total;

}

Future<void> removeFromCart(String productId) async {

var removedItem = \_cartItems.firstWhere(

(item) => item['productId'] == productId,

orElse: () => {},

);

if (removedItem.isNotEmpty) {

await \_updateStock(productId, removedItem['quantity'], true);

\_cartItems.remove(removedItem);

notifyListeners();

await \_updateFirestoreCart();

}

}

Future<void> clearCart() async {

\_cartItems.clear();

notifyListeners();

await \_updateFirestoreCart();

}

Future<void> \_updateFirestoreCart() async {

final userCartCollection = FirebaseFirestore.instance.collection('cart').doc(\_userId);

await userCartCollection.set({'cartItems': \_cartItems});

}

int productCount(String productId) {

return \_cartItems.where((item) => item['productId'] == productId).length;

}

List<Map<String, dynamic>> get uniqueProducts {

Set<String> uniqueProductIds = Set<String>();

List<Map<String, dynamic>> uniqueProductsList = [];

for (var item in \_cartItems) {

if (uniqueProductIds.add(item['productId'])) {

uniqueProductsList.add(item);

}

}

return uniqueProductsList;

}

int get uniqueProductCount {

return uniqueProducts.length;

}

void decreaseQuantity(String productId) async {

var productIndex = \_cartItems.indexWhere((item) => item['productId'] == productId);

if (productIndex != -1) {

\_cartItems[productIndex]['quantity'] = (\_cartItems[productIndex]['quantity'] ?? 1) - 1;

if (\_cartItems[productIndex]['quantity'] == 0) {

await removeFromCart(productId);

} else {

await \_updateStock(productId, 1, true);

notifyListeners();

await \_updateFirestoreCart();

}

}

}

void increaseQuantity(String productId) async {

var productIndex = \_cartItems.indexWhere((item) => item['productId'] == productId);

if (productIndex != -1) {

\_cartItems[productIndex]['quantity'] = (\_cartItems[productIndex]['quantity'] ?? 0) + 1;

await \_updateStock(productId, 1, false);

notifyListeners();

await \_updateFirestoreCart();

} else {

addToCart({

'productId': productId,

'quantity': 1,

});

notifyListeners();

}

}

Future<void> initializeCartFromFirestore() async {

await fetchCartFromFirestore();

}

Future<void> \_updateStock(String productId, int quantity, bool increase) async {

final productCollection = FirebaseFirestore.instance.collection('products');

final productData = await productCollection.doc(productId).get();

int productStock = productData['stock'] ?? 0;

if (increase) {

productCollection.doc(productId).update({

'stock': productStock + quantity,

});

} else {

productCollection.doc(productId).update({

'stock': productStock - quantity,

});

}

}

}

**Payment – paymentService.dart**

import 'package:mailer/mailer.dart';

import 'package:mailer/smtp\_server/gmail.dart';

import 'package:cloud\_firestore/cloud\_firestore.dart';

import 'package:farmconnect/features/…/Cart/cartProvider.dart';

import 'package:farmconnect/features/…/Cart/orderConfirmationPage.dart';

import 'package:firebase\_auth/firebase\_auth.dart';

import 'package:flutter/material.dart';

import 'package:provider/provider.dart';

import 'package:razorpay\_flutter/razorpay\_flutter.dart';

class PaymentService {

static void handlePaymentSuccess(BuildContext context, PaymentSuccessResponse response) async {

await \_storePaymentDetails(context, response.paymentId!);

\_sendPaymentNotificationEmail(context);

\_navigateToOrderConfirmation(context, response.paymentId!);

}

static Future<void> \_storePaymentDetails(BuildContext context, String paymentId) async {

final cartProvider = Provider.of<CartProvider>(context, listen: false);

double totalAmount = cartProvider.totalAmount() ?? 0.0;

try {

User? user = FirebaseAuth.instance.currentUser;

String userUid = user?.uid ?? '';

String customerName = user?.displayName ?? 'Guest';

String customerEmail = user?.email ?? 'guest@example.com';

String customerPhone = user?.phoneNumber ?? 'N/A';

List<Map<String, dynamic>> productsList = cartProvider.cartItems.map((item) {

double productPrice = item['productPrice'] ?? 0.0;

double totalPrice = productPrice \* item['quantity'];

return {

'productId': item['productId'],

'productName': item['productName'],

'quantity': item['quantity'],

'unitPrice': productPrice,

'totalPrice': totalPrice,

};

}).toList();

await FirebaseFirestore.instance.collection('payments').add({

'paymentId': paymentId,

'amount': totalAmount,

'timestamp': FieldValue.serverTimestamp(),

'userUid': userUid,

'customerName': customerName,

'customerEmail': customerEmail,

'customerPhone': customerPhone,

'products': productsList,

});

cartProvider.clearCart();

} catch (e) {

print("Error storing payment details: $e");

}

}

static void handlePaymentError(PaymentFailureResponse response) {

print("Payment error: ${response.message}");

}

static void handleExternalWallet(ExternalWalletResponse response) {

print("External wallet: ${response.walletName}");

}

static void \_navigateToOrderConfirmation(BuildContext context, String paymentId) {

Navigator.pushReplacement(

context,

MaterialPageRoute(

builder: (context) => OrderConfirmationPage(paymentId: paymentId),

),

);

}

static void \_sendPaymentNotificationEmail(BuildContext context) {

final cartProvider = Provider.of<CartProvider>(context, listen: false);

User? user = FirebaseAuth.instance.currentUser;

String customerEmail = user?.email ?? 'guest@example.com';

try {

sendNotificationEmail(customerEmail, true);

} catch (e) {

print("Error sending payment notification email: $e");

}

}

static void sendNotificationEmail(String recipient, bool isActive) async {

final smtpServer = gmail('namalthomson2024b@mca.ajce.in', 'Amal664422');

final message = Message()

..from = Address('admin@farmconnect.com', 'Admin FarmConnect')

..recipients.add(recipient)

..subject = 'Payment Successful and Order Placed'

..html = '''

<html>

<head>

<style>

body {

font-family: 'Helvetica Neue', Arial, sans-serif;

background-color: #f9f9f9;

color: #333;

margin: 0;

padding: 0;

}

.container {

max-width: 600px;

margin: 0 auto;

border-radius: 10px;

overflow: hidden;

box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);

}

.header {

color: #fff;

text-align: center;

padding: 20px;

}

h1 {

color: #fff;

}

.content {

padding: 30px;

background-color: #ffffff; /\* White \*/

}

p {

line-height: 1.6;

color: #333; /\* Dark gray for better visibility on white \*/

}

.footer {

background-color: #f9f9f9;

padding: 20px;

text-align: center;

color: #888;

font-style: italic;

}

</style>

</head>

<body>

<div class="container">

<div class="header">

<h1>Payment Successful and Order Placed</h1>

</div>

<div class="content">

<p>Dear Customer,</p>

<p>Your payment was successful, and your order has been placed.</p>

<p>Thank you for shopping with us!</p>

</div>

<div class="footer">Best regards, Admin FarmConnect</div>

</div>

</body>

</html>

''';

try {

final sendReport = await send(message, smtpServer);

print('Message sent: ${sendReport}');

} on MailerException catch (e) {

print('Message not sent. ${e.message}');

}

}

static void startPayment(BuildContext context) {

final cartProvider = Provider.of<CartProvider>(context, listen: false);

double totalAmount = cartProvider.totalAmount() ?? 0.0;

var options = {

'key': 'rzp\_test\_x1IywbsCJ3R5CZ',

'amount': (totalAmount \* 100).toInt(),

'name': 'FarmConnect',

'description': 'Payment for your order',

'prefill': {

'contact': '9469664422',

'email': 'farmconnectadm@gmail.com',

},

'external': {

'wallets': ['paytm'],

},

};

try {

Razorpay \_razorpay = Razorpay();

\_razorpay.on(Razorpay.EVENT\_PAYMENT\_SUCCESS, (response) {

handlePaymentSuccess(context, response);

});

\_razorpay.on(Razorpay.EVENT\_PAYMENT\_ERROR, (response) {

handlePaymentError(response);

});

\_razorpay.on(Razorpay.EVENT\_EXTERNAL\_WALLET, (response) {

handleExternalWallet(response);

});

\_razorpay.open(options);

} catch (e) {

print("Error in starting Razorpay payment: $e");

}

}

}

## Screen Shots

|  |  |  |
| --- | --- | --- |
|  |  |  |
| *9.1.1 Splash Screen* | *9.1.2 Login Page* | *9.1.3 SignUp Page* |
|  |  |  |
|  |  |  |
|  |  |  |
| *9.1.4 SignIn with Google* | *9.1.5 Forget Password* | *9.1.6 Terms & Conditions* |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| *9.1.7 Admin Dashboard* | *9.1.8 Farmer Details* | *9.1.9 Buyer Details* |
|  |  |  |
|  |  |  |
|  |  |  |
| *9.1.10 Pending Approval* | *9.1.11 Approved Products* | *9.1.12 Payments* |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| *9.1.13 Order Details* | *9.1.14 Add Products* | *9.1.15 My Products* |
|  |  |  |
|  |  |  |
|  |  |  |
| *9.1.16 Products Listing* | *9.1.17 Cart Page* | *9.1.18 My Profile* |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| *9.1.19 Update Password* | *9.1.20 Update Profile* | *9.1.21 Order Details* |
|  |  |  |
|  |  |  |
|  |  |  |
| *9.1.22 Bill Generation* |  |  |