# Enhancing Productivity in Sri Lanka's Agriculture Sector with a Cloud-Based System for Data Acquisition and Representation to Facilitate Informed Decision Making

**L.R.S.D. Rathnayake**

**2024**

# Enhancing Productivity in Sri Lanka's Agriculture Sector with a Cloud-Based System for Data Acquisition and Representation to Facilitate Informed Decision Making

**A dissertation submitted for the Degree of Master of Information Technology**

**L.R.S.D. Rathnayake**
**University of Colombo School of Computing**
**2024**

# Declaration

| |
|---|
| **Name of the student:** L.R.S.D. Rathnayake |
| **Registration number:** 2020/MIT/083 |
| **Name of the Degree Programme:** Master of Information Technology |
| **Project/Thesis title:** Enhancing Productivity in Sri Lanka's Agriculture Sector with a Cloud-Based System for Data Acquisition and Representation to Facilitate Informed Decision making |

1. The project/thesis is my original work and has not been submitted previously for a degree at this or any other University/Institute. To the best of my knowledge, it does not contain any material published or written by another person, except as acknowledged in the text.

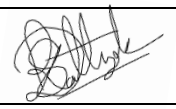
2. I understand what plagiarism is, the various types of plagiarism, how to avoid it, what my resources are, who can help me if I am unsure about a research or plagiarism issue, as well as what the consequences are at University of Colombo School of Computing (UCSC) for plagiarism.

3. I understand that ignorance is not an excuse for plagiarism and that I am responsible for clarifying, asking questions and utilizing all available resources in order to educate myself and prevent myself from plagiarizing.

4. I am also aware of the dangers of using online plagiarism checkers and sites that offer essays for sale. I understand that if I use these resources, I am solely responsible for the consequences of my actions.

5. I assure that any work I submit with my name on it will reflect my own ideas and effort. I will properly cite all material that is not my own.

6. I understand that there is no acceptable excuse for committing plagiarism and that doing so is a violation of the Student Code of Conduct.

| **Signature of the Student** | **Date (DD/MM/YYYY)** |
|---|---|
|  | 25.09.2024 |

---

Certified by Supervisor(s)

This is to certify that this project/thesis is based on the work of the above-mentioned student under my/our supervision. The thesis has been prepared according to the format stipulated and is of an acceptable standard.

| | **Supervisor 1** | **Supervisor 2** | **Supervisor 3** |
|---|---|---|---|
| **Name** | Prof. M.G.N.A.S. Fernando | | |
| **Signature** |  | | |
| **Date** | 24.09.2024 | | |

# Acknowledgements

I extend my sincere gratitude to Prof. M.A.G.N.A.S. Fernando for his exceptional supervision and unwavering support throughout every stage of this project's journey to successful completion. His invaluable guidance and expertise played a pivotal role in ensuring its success.

I am deeply thankful to the University of Colombo School of Computing for providing me with a remarkable opportunity and a robust platform to enhance my knowledge and skills in Information Technology through its esteemed postgraduate degree program.

My heartfelt appreciation goes out to all the lecturers, visiting scholars, and non-academic staff at UCSC who generously invested their time and efforts to equip us with the essential knowledge, skills, and IT-centric mindset vital for success in the field of Information Technology.

The invaluable assistance and guidance of numerous respected professionals significantly contributed to the accomplishment of this project. Lastly, I express profound gratitude to my husband, family, and colleagues for their unwavering support and encouragement. Without their inspiration, this challenging endeavor would not have been possible.

# Abstract

The agricultural sector is a major component in Sri Lanka economy, employing a significant portion of country's population and under the guidance and support of the department of agriculture. Agriculture sector struggles due to unreliable decision-making, caused by a lack of local data recording. This issue hampers effective management and strategy implementation within the sector by today.

This project introduces "Ceylon AgriData", a cloud-based system designed to revolutionize the agricultural sector in Sri Lanka. "Ceylon AgriData", aims to enhance operational efficiency by transitioning from traditional paper-based data collection methods to a digital, cloud-based approach. This transition is facilitated through a mobile application tailored for agricultural officers, enabling streamlined data collection and management directly from the field.

By digitizing data collection, "Ceylon AgriData" significantly enhances data accuracy and accessibility, storing it securely on a centralized cloud database. This database is accessible to a wide range of stakeholders, including department administrators, field officers, researchers, and other key stakeholders through user-friendly interfaces. The system's web application features intuitive dashboards and tools, empowering stakeholders to make well-informed decisions, develop effective policies, and provide superior support to farmers, thus promoting sector-wide efficiency.

Besides, "Ceylon AgriData" encompasses a comprehensive webapp that allows for the generation of detailed reports from the aggregated agricultural data, facilitating streamlined operations and informed decision-making within the sector. A message broadcasting service is also integrated, ensuring the timely dissemination of important information to enhance information sharing and engagement across the sector. The system incorporates a free advertising service, enabling farmers to directly market their products without intermediaries. This feature not only facilitates better sales opportunities for farmers but also aids the government in making informed decisions on price regulation. By analyzing the recorded data on crop yields, products, and prices, the government can implement policies that ensure fair pricing and market stability.

The development of "Ceylon AgriData" utilized a robust technology stack, including Python and Flask for backend services, JavaScript and React for the web platform, and the Flutter framework for mobile application development. This combination has resulted in the creation of a user-centric mobile app and a web-based system, both integrated via a REST API service.

An extensive testing workflow was employed to validate the system's functionality, encompassing unit, integration, and user acceptance testing phases. These tests, conducted both manually and automatically, highlighted the system's usability and identified areas for further refinement.

The project was managed using the iterative waterfall model, ensuring structured progress and adaptability throughout its development. This methodical approach guaranteed that "Ceylon AgriData" not only met its initial objectives but also laid a foundation for ongoing enhancements, setting a new standard for technological innovation in Sri Lanka's agricultural sector.

# Table of Contents

# List of Figures

# List of Tables

# List of Acronyms

REST API - Representational State Transfer Application Programming Interface

ORM - Object-Relational Mapping

CLI - Command Line Interface

NPM - Node Package Manager

CORS - Cross-Origin Resource Sharing

JSON- JavaScript Object Notation

DOM- Document Object Model

# Chapter 1 - Introduction

The agriculture sector is important to Sri Lanka's economy and especially, most of the rural population employs in farming. In 2021, agriculture contributed 8.7% to the country's GDP(Aaron O'Neill, 2023). Sri Lanka primarily produces rice, vegetables, and fruits for domestic consumption, while also exporting commodities such as tea, rubber, and coconut. Many people in the country rely on agriculture as their primary occupation.

## 1.1 Project Overview

The project objective is to increase efficiency in Sri Lanka's agriculture sector by introducing a cloud-based system, *"Ceylon AgriData"* for collecting and representing up-to-date agriculture data. This system will enable informed decision-making in the agriculture sector and improve its services.

A user-friendly mobile application will be implemented for agricultural field officers who usually responsible in collecting farmer related data during field visits. This application will enable them to effortlessly gather crucial data such as farmer details, farm details such that locations, cultivating crop categories, yields, costs, pesticide and fertilizer usage, as well as occurring crop disease and disaster details, if any. This will replace the current manual, paper-based method of agricultural data collection with the proposed mobile and web applications, improving data accuracy and accessibility.

The collected data will be securely stored in a centralized cloud database, accessible to relevant stakeholders such as agricultural department administrators, agricultural field officers, researchers, and other key players in the agricultural sector. The cloud platform will provide user-friendly dashboards, report generating functionalities, user and agriculture data management services assisting stakeholders to make informed decisions and develop policies to support all stakeholders in agricultural sector.

Additionally, the project aims to launch a free advertising service within the system, assisting farmers to expose to local and foreign markets. It will enable farmers to enter product details and find potential buyers, ensuring fair pricing and better selling opportunities freely. Thereby, government will have oversight in regulating prices and promoting fair market practices.

Furthermore, the project will establish a communication service to facilitate timely information dissemination (meetings, notices of distributing fertilizers, etc.) among agricultural officers, registered farmers. This will ensure the efficient broadcasting of messages, keeping all relevant parties informed about important updates and initiatives.

By implementing this cloud-based system, “Ceylon AgriData”, aims to provide up-to-date and comprehensive agricultural data, empowering stakeholders to make better decisions, support farmers, and enhance overall efficiency in Sri Lanka's agriculture sector.

## 1.2 Background of Study

Agriculture plays a vital role in the country's economy, contributing to the GDP and providing employment opportunities, particularly in rural areas. Currently, the agriculture sector is undergoing various challenges such that a lack of knowledge on sustainable agriculture, lack of financial support, lack of a proper management system, and proper ways to share knowledge and information among agricultural parties, as a result, having low productivity and value for their harvest can be stated. A key problem is the lack of detailed, trustworthy data, making it hard to make informed decisions. Therefore, even though agriculture is a significant source of economic growth in Sri Lanka, the sector's productivity remains low at the moment.

The Sri Lankan agriculture sector, heavily relies on government support, including free irrigation and extension services, substantial fertilizer subsidies, support prices, and trade protection measures (Manoj Thibbotuwawa, 2021).The lack of updated agricultural data has hindered the government's efforts to enhance efficiency and productivity in the agriculture sector. Without access to necessary and current information, it becomes challenging for the government to implement effective measures to improve agricultural productivity.

One of the key challenges is the reliance on traditional and manual data collection methods. Agricultural officers often rely on paper-based systems to gather information, which is time-consuming, prone to errors and lacks real-time accessibility. This inefficiency in data collection limits the ability to make informed decisions and develop effective policies to support the agriculture sector

Many farmers in Sri Lanka face difficulties accessing market opportunities and receiving reasonable product prices. Limited information and inadequate platforms for connecting farmers

with buyers result in market inefficiencies and reduced profitability, especially when they lack awareness of the specific prices in the Colombo market.

Furthermore, there is a need for improved communication between agricultural departments, officers, and farmers. Timely dissemination of information, updates, and important notifications is crucial for effective decision-making and addressing challenges promptly.

The background of this study recognizes these challenges and aims to address them by proposing the development of a comprehensive system consisting of a mobile application and a cloud-based platform. This system will streamline data collection processes, provide real-time information access, establish a marketplace for farmers, and enhance communication between stakeholders.

## 1.3 Motivation

The motivation for this project is to address challenges and unlock the potential in Sri Lanka's agriculture sector, which plays a vital role in the economy but faces obstacles to its growth and development.

One key motivation is the need for improved data acquisition and management. Traditional manual methods of data collection in agriculture are time-consuming, error-prone, and lack real-time accessibility. By developing a mobile application and a cloud-based platform, this project aims to streamline the data collection process, enhance accuracy, and provide stakeholders with timely and comprehensive information. This will enable informed decision-making, effective resource allocation, and the formulation of evidence-based policies to support the agriculture sector.

Another motivation is to address the market inefficiencies faced by farmers. Limited access to market opportunities and a lack of information regarding fair prices undermine the profitability of farmers. By establishing a free advertising service within the proposed system, this project seeks to assist farmers to connect directly with buyers, ensuring efficiency in sales and enhancing market transparency. The integration of government oversight will further enable price regulation and market regulation, creating a favorable environment for farmers.

Timely and accurate communication is vital for addressing challenges promptly, providing essential updates, and facilitating effective collaboration. The “Ceylon AgriData” system, will

serve as a communication tool, enabling the broadcasting of important messages and notifications to relevant stakeholders, ultimately improving overall efficiency and productivity.

Overall, the motivation behind this thesis is to leverage technology and innovation to overcome the limitations and inefficiencies within Sri Lanka's agriculture sector. By addressing data collection, market access, and communication challenges, this study aims to enhance productivity, promote sustainable agricultural practices, and ultimately contribute to the socio-economic development of the country.

## 1.4 Objectives

1. To design and develop a mobile application system for agricultural field officers to collect farmer related data during field visits.

   This mobile system would assist regional agricultural field officers to collect essential agricultural data such as details of farmers, farms, and its locations, encapsulated crop categories, yields, costs, consumption of pesticides and fertilizers, and crop disease and disaster details. The system will provide a more efficient data acquisition method compared to the current manual process that relies on paper documentation. This mobile system will improve data accuracy and accessibility, enabling agricultural departments to make informed decisions and enhance overall agricultural services

2. To develop a cloud-based system to store, maintain and represent agricultural data.

   This will enable easy access to and sharing of information for various stakeholders, including agriculture departments, scientists, researchers, farmers, and other agricultural stakeholders. The platform aims to facilitate the stakeholders with accurate and timely data that can be used to make better decisions to support the country's agricultural sector.

3. Launch a free advertising service to help farmers access better market opportunities and sell their products at favorable prices. This service will also enable exporters to buy directly from farmers, improving the efficiency of sales and purchases. Additionally, the government will monitor pricing to ensure fairness.

4. To provide efficient and on-time message conveying facilities from the Agriculture Department to regional agricultural officers, and farmers.

## 1.5 Scope of Study

The project's initial phase will focus on implementing the system as per the requirements in the government agricultural regional office located in Dodangoda, Kalutara District, Sri Lanka.

The proposed project aims to enhance the agriculture sector in Sri Lanka through the development of a comprehensive system consisting of a mobile app and a cloud-based platform. The mobile app will replace traditional paper-based methods, allowing regional agricultural officers to efficiently and accurately collect agriculture-related data in the field. This includes farmer information, farm and cultivation details, aid distribution records, and disaster reports, streamlining the data collection process for improved decision-making. The project also includes the implementation of a field mapping facility, allowing agriculture field officers to digitally enter field locations ensuring their field visits more transparent and efficient. The mobile application will facilitate effective communication among the agriculture department, field agriculture officers and farmers. Important information and updates can be broadcasted through the application, ensuring that officers have the most up-to-date information to respond promptly. Also, the app will assist in gathering data on crop damages in times of disasters such that flooding, crop diseases etc., enabling the allocation of resources and support to affected farmers in a timely manner. Furthermore, the application will collect farmer details, which can be utilized to inform them about fertilizer distribution, donations, and other forms of assistance through broadcasting service.

All collected data will be securely stored in a centralized cloud database, accessible to agricultural department administrators, researchers, and other stakeholders through user-friendly dashboards. This database will serve as a valuable resource for making informed decisions and developing policies to support farmers. The database will provide up-to-date data for analysis, enabling stakeholders to address agricultural challenges by identifying trends, patterns, and potential solutions.

The project will also establish a free advertising service where farmers can conveniently enter the details of their yields and connect with potential buyers. The goal is to accommodate farmers with limited computer literacy by offering a user-friendly interface. By facilitating direct communication between farmers and buyers, the marketplace promotes transparency and efficiency in the agricultural supply chain.

The platform will enable the government to monitor pricing trends, allowing for real-time regulation of prices and promotion of fair market practices under the supervision of agricultural administrators. Government participation, especially for farmers registered with regional agricultural offices, will facilitate timely interventions and support. This government-supervised free advertising portal will ensure that farmers can sell their products at fair prices and receive the assistance they need from authorities, creating a supportive environment for agricultural commerce.

## 1.6 Structure of the Dissertation

Chapter 2, the Literature Review and Similar Systems examines the literature review (similar systems) related to the agriculture sector in Sri Lanka, focusing on challenges and the importance of data acquisition and informed decision-making. In Chapter 3, the Design chapter, the system architecture, data acquisition methods, storage and processing techniques, and user interface design are discussed. Chapter 4, Implementation, details the steps taken to develop and implement the cloud-based system, including software and hardware components, programming languages, and any challenges encountered. Chapter 5, Testing and Evaluation, covers the testing procedures, results, and any identified limitations or areas for improvement. The dissertation concludes with the Conclusion, summarizing key findings, discussing implications, and suggesting future research directions. Also. the thesis includes a references section and appendices that provide supplementary information, data, or materials that support and enhance the project.

# Chapter 2 – Background

## 2.1 Introduction

Agriculture serves as the foundational sector in a majority of nations across the globe. China, India, Japan, Mexico, Brazil, Russia, USA, and France stand out as prominent examples of countries that display great potential in this regard (FAO, 2023). According to the World Bank, the advancement of agriculture emerges as a highly potent instrument for eradicating extreme poverty, promoting inclusive economic growth, and ensuring sustenance for an estimated 9.7 billion individuals by the year 2050 (WorldBank, 2023). To achieve specified objectives, people have adopted automation, precision farming, and smart agricultural practices utilizing information technology to advance their agricultural sectors and attain desired results (Jayathilake et al., 2010). However, a comprehensive data collection center is essential to increase the productivity of agriculture.

Sri Lanka's economy heavily depends on agriculture, a sector pivotal for its development, employment generation, and food security for all. It significantly contributes to the nation's prosperity and sustains livelihoods, particularly in rural regions (Ministry of Agriculture, 2023). Agriculture in Sri Lanka encompasses crop cultivation, animal husbandry, fishing, and forestry.

However, the agriculture sector in Sri Lanka faces challenges due to limited access to timely and accurate agricultural data. This lack of information hampers decision-making and prevents stakeholders from improving agricultural practices and resource allocation. Traditional data collection methods, such as manual surveys and paper records, are slow, labor-intensive, and vulnerable to errors, causing delays in analysis and response. Inefficient data representation further complicates the integration and utilization of valuable insights. To address these challenges, a centralized crop data management platform is needed. This platform would assist administrators collecting, storing, analyzing and representation of agricultural data and farmers to access agricultural information and market opportunities by directly finding vendors from the system. It would also support decision-making and scenario planning in the agricultural sector

## 2.2 Literature Review and Similar Systems

The agricultural sector has significantly adopted information technology (IT) to enhance the effectiveness and productivity of farming practices. The latest advancements in IT have introduced smart devices, sensor technologies, web services and applications, which offer immense potential to promote sustainability and productivity in the agriculture industry (Cheema and Khan, 2019). Precision farming and e-farming techniques have simplified agricultural processes, while the availability of online databases containing comprehensive agricultural data has created a valuable platform for policymakers and researchers to actively contribute to the improvement of the agricultural sector worldwide.

Prominent cloud-based agricultural management software systems, such as 'Bushel', 'Granular', 'FieldView', 'AgriWebb', 'AgSense', 'Conservis', 'Agrible-Morning Farm Report', 'aWhere' and 'Cropin' (Agrible-Morning Farm Report, AgriWebb, AgSense, 2003, aWhere, BushelFarm, Climate Corporation, Conservis, 2008, Cropin, 2010, Granular) are widely utilized in modern farming practices to optimize and streamline operations. Also, they are utilized with crop monitoring, analytics, crop information and advisory features. These systems offer a diverse range of applications and advantages, encompassing crop and field management, inventory and supply chain management, data analytics and decision support, financial management, compliance and reporting, as well as remote monitoring and control. By integrating advanced technology and data-driven insights, farmers can effectively enhance the efficiency, sustainability, and profitability of their agricultural practices. Overall, these software applications can help farmers to streamline their operations, optimize resource utilization, and improve their yields and profitability.

'AgriWebb' (AgriWebb) is a simple application that can be used for global cattle and sheep production provenance, profitability and sustainability across the supply chain and it offers a user-friendly interface that can be easily accessed and utilized by farmers, along with real-time data entry and insights into crop management processes. One disadvantage of the 'AgriWebb' (AgriWebb) is that it requires a stable internet connection, which may be a challenge for farmers working in rural areas with limited connectivity. 'Farmforce' (Farm Force) is a cloud-based crop management system, developed in Kenya. The system offers tools for tracking crop activities, monitoring yields and sales, and analyzing data to improve farm performance.

Various system applications have been identified in the literature as being utilized in Sri Lanka for agricultural purposes, aiming to enhance relative productivity. These applications include 'Cropin',

'Agri Manager', 'Krushi Advisor', 'Govi Mithuru', 'Helawiru', 'Coconut App', 'MyAgri' and ‘Agri Life’ (Agrimanager, Coconut Cultivation Board, 2023, Cropin, 2010, Department of Agriculture, 2021, Govi Mithuru, 2015, Helawiru, KOMA Labs, 2020, My Agri, 2021) . These applications are utilized to enhance the relative productivity of agriculture in Sri Lanka. ‘Coconut App’ is an Android application that is facilitated with services to establish a sustainable coconut cultivation by providing necessary input and financial facilities to coconut growers island-wide through an efficient extension & advisory service (Coconut Cultivation Board, 2023). ‘Agro life’ (KOMA Labs, 2020) is a mobile app designed for Sri Lankan farmers and agricultural community which provides complete information on crop production, crop protection, fertilizers, machinery, and impact of climate, storage procedures and all relevant allied services.

The mobile-based Android application systems, such that, 'Krushi Advisor', 'Govi mithuru', and 'My Agri' (Department of Agriculture, 2021, Govi Mithuru, 2015, My Agri, 2021) serve as comprehensive advisory platforms for farmers and relevant stakeholders, providing them with self-guidance regarding crops, seeds, cultivation information, and appropriate fertilizers. By leveraging these applications, farmers can access valuable knowledge and become more informed in their agricultural practices.

'Helaviru', and 'Agrithing' (Agrithing, Helawiru) are dynamic digital marketplaces that facilitate trading activities of agriculture-related products and harvests for small-scale collectors, retailers, and large consumers. These platforms were developed to enable timely sales of products at reasonable prices, thereby benefiting both sellers and buyers in the agricultural market. However, the lack of maintenance, reliability have made these systems low productive and disqualified to be used in accurate decision making by the government at the moment.

Table 2.1 provides a summarized comparison of commonly used agricultural aid software applications in Sri Lanka. The table highlights that most of these applications are primarily focused on knowledge delivery, offering information and guidance to farmers. Additionally, some applications also provide real-time market information, allowing farmers to stay updated on pricing trends and market conditions. Unfortunately, a limited number of agricultural aid mobile applications are presently operational and functioning effectively in Sri Lanka. Consequently, this situation has resulted in inefficiencies, inadequate tracking, and reduced productivity within the agricultural sector of the country. However, considering the related works it clearly shows the necessity of a system that provide accurate and updated information

among the farmers, traders and the department of agriculture for informed decision making to improve the agriculture sector in Sri Lanka especially as a developing country amidst the economic crisis. As a developing country, Sri Lanka is in need of an intelligent centralized system for agricultural data and a mechanism for informed decision-making to enhance the efficiency and productivity of its agricultural sector. The government has been actively supporting the agricultural sector by providing substantial freely available facilities to farmers, distinguishing Sri Lanka from other countries worldwide. Consequently, it is crucial for the government to establish a mechanism that enables accurate insights of the current market information of agricultural products for regulatory purposes and decision making.

## 2.3 Related Technologies

Understanding the underlying technologies is essential for comprehending the functionalities and capabilities of these applications in supporting the agricultural sector. The aforementioned technologies, namely cloud computing, sensor technologies, mobile application development, data analytics, machine learning, database management systems, geospatial technologies, and communication protocols, can be primarily classified as related technologies utilized in agriculture management systems (Christine Zhenwei Qiang et al., 2012, Ojha et al., 2015, Rathod et al., 2022).

Cloud computing enables the storage, management, and processing of agricultural data in cloud-based platforms, allowing for scalable and accessible storage and collaboration among stakeholders (Singh et al., 2020). Mobile applications are created specifically for agriculture management systems to offer farmers convenient access to information, advisory services, market updates, and communication channels while on the move (Jayathilake et al., 2010).

Combination of cloud-based software, mobile application development, communication protocols, and database management systems forms a comprehensive technological framework for effective agricultural software services. Cloud-based software provides a scalable and accessible platform for agricultural data management, allowing for efficient storage, processing, and collaboration among stakeholders. This technology enables the centralization of agricultural data, ensuring its availability and security. Mobile application development is another significant technology that empowers farmers by offering on-the-go access to vital information, advisory services, market updates, and communication channels. These applications facilitate real-time decision-making, enabling farmers to access relevant data and services directly from their mobile devices,

irrespective of their location. Communication protocols are essential for establishing secure and efficient communication channels within agricultural management systems. These protocols enable seamless data sharing, collaboration, and synchronization between different components of the system. Real-time data exchange ensures timely and accurate information dissemination, enhancing coordination and decision-making processes. Database management systems (DBMS) are critical for efficient organization, storage, and retrieval of agricultural data. These systems ensure data integrity, provide robust querying capabilities, and enable seamless access to relevant information.

## 2.4 Existing System Processes and Functionalities

Existing system, 'Agrimanager'(Agrimanager) closely aligns with the proposed system's requirements and functionalities. User management, data collection, data storage and retrieval, communication, market connectivity, and system administration, closely match the desired features of the proposed system. By leveraging its capabilities, the new system can benefit from an established framework and effectively incorporate and enhance existing functionalities. Figure 2.1 illustrates the top-level use case diagram of the existing system, showcasing its functionalities and processes that align with the proposed system. The diagram presents an overview of the key interactions and relationships between actors and use cases within the existing system, highlighting the core functionalities and their corresponding actors.

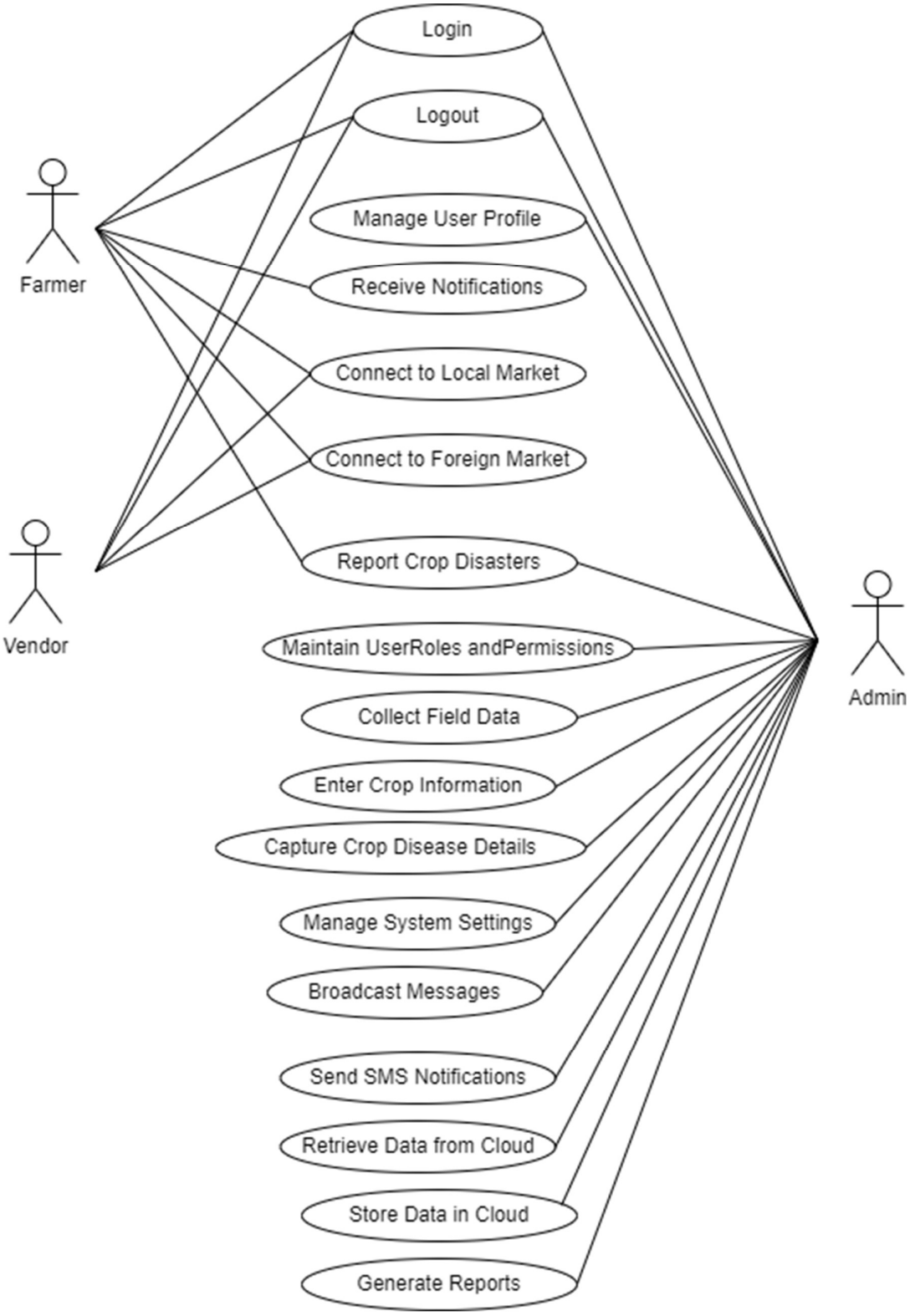



*Figure 2.1 : Top level use case diagram of the system, "Agrimanager" (Agrimanager)*

| Application Name | Technologies Used | Features | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Crop Guidance | Analysis & Reporting | Fertilizers | Pest & Diseases | Weather Forecasts | Marketplace Information | News/Alerts | Other |
| Krushi Advisor | Mobile Application Development (Android) | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | Cultivation Techniques<br>Access to government schemes |
| Govi Mithuru | Mobile Application Development (Android)<br>Cloud Storage and Retrieval | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | Expertise's Advices<br>Access to government schemes |
| Agri Manager | Web-based platform development<br>Cloud storage and retrieval Data Analytics | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | Inventory Management<br>Financial Management<br>Task & Labor Management<br>Suitable for Small, Mid-size businesses, Large enterprises |
| Coconut App | Mobile Application Development (Android)<br>Cloud Storage and Retrieval | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | Cultivation Techniques<br>Access to government schemes |
| Helaviru | Mobile Application Development (Android, iOS)<br>Cloud Storage and Retrieval | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | Real time Market Information |
| AgriCloud (Proposed System) | Mobile Application Development (Android)<br>Web Application<br>Cloud Storage and Retrieval | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | Cultivation Techniques<br>Up to date Market Information<br>Field Map Tracking for officers<br>Publishing Adverticement |

*Table 2.1 : Summarized comparison of commonly used agricultural aid software applications in Sri Lanka*

## 2.4 Requirement Analysis

### 2.4.1 Introduction

Requirement analysis provides a clear understanding of user needs, enabling the development of a system that effectively addresses the requirements of agricultural officers, farmers, administrators, and other stakeholders. The analysis helps in designing an intuitive and user-friendly mobile application and a robust cloud-based platform, ensuring the system meets the specific needs of the agriculture sector in Sri Lanka. Accurate resource estimation and scope management keep the project on track and prevent budget and timeline issues. Identifying risks and challenges early on helps us address and minimize their impact on the project. When stakeholders are aligned, their expectations are taken into account, resulting in higher satisfaction. Additionally, the analysis enables us to collect, store, and manage data efficiently, which supports making well-informed decisions. Thus, it defines the design and development of a comprehensive cloud–based system for data acquisition and representation to facilitate informed decision-making.

### 2.4.2 Overall Description

1. Product Perspective

The proposed system, “Ceylon AgriData” will be an implemented software solution that includes third-party libraries and API services. For more details, see Chapters 4.

2. Product Features

The main features and functionalities of the system, including:

- Mobile application for agricultural field officers to collect farmer related data efficiently during field visits
- Cloud-based system for securely storing and maintaining agricultural data efficiently
- Data representation via different dashboards to support stakeholders for informed decision-making
- Broadcast message service among agriculture stakeholders
- A complimentary advertising platform designed to directly connect farmers with vendors, eliminating intermediaries, while enabling government agencies to monitor and regulate pricing effectively.

3. User Classes and Characteristics

The system will mainly cater to the following user classes:

- Agricultural Officers: Responsible for collecting farmer data using the mobile application and will benefit in making reports.
- Farmers: Users who will benefit from the complementary advertising platform service and receive broadcast messages on a timely manner.
- Administrators: Manage and maintain the system, including access control and will benefit in generating reports.
- Researchers: Access agricultural up-to-date data to analyze data for research purposes.

4. Operating Environment

The system should be compatible with the following operating environment:

- Mobile Application: Android devices with the latest versions.
- Cloud Platform: Web browsers and internet connectivity

5. Design and Implementation Constraints

- The system should be user-friendly, ensuring ease of use for agricultural officers and farmers with varying technical expertise.
- The cloud-based platform should be capable to handle a large volume of agricultural data and concurrent user access.

6. Assumptions and Dependencies

- It is assumed that mobile devices and internet connectivity will be available to agricultural officers and farmers at least for a limited time period regularly in the target areas.
- The system will depend on reliable network connectivity to synchronize data between the mobile application and the cloud platform.
- The availability and accuracy of agricultural data depend on the timely input and cooperation of agricultural officers.

### 2.4.3 Functional Requirements

1. User Authentication and Authorization

User authentication and authorization of the system will be handled by basic authorizations (using user name and password); encrypted them and Bearer token to maintaining sessions.

- **Login/Logout:** Users, including agricultural officers, administrators, and farmers, must securely log in and out of the system using username and password credentials.
- **Role-Based Access Control:** Depending on the user's role (e.g., regional agricultural officer, admin, farmer), the system will grant different access levels, ensuring that sensitive operations (such as data modification) are restricted to authorized personnel.
- **Session Management:** Sessions are maintained using encrypted bearer tokens, ensuring continuous security without requiring frequent reauthentication.

2. Data Management

The software enables the storage, retrieval, and management of agricultural data in a cloud database. It supports seamless integration with various data sources, ensuring data integrity, reliability, and scalability. The cloud-based system provides secure storage for the collected agricultural data.

**Cloud Database Integration:** The system integrates with a cloud-based database, ensuring secure and scalable data storage for large volumes of agricultural data, including farmer records, cultivation data, and disaster reports.

- **Data Storage:** All data collected through the mobile app and web app will be securely stored in the cloud, with redundancy measures for data recovery in case of system failures.
- **Data Retrieval and Syncing:** Users can retrieve and synchronize data in real-time between the mobile app, web app and the cloud. The system automatically resolves conflicts when data is updated simultaneously by different users.

3. Data Collection

The mobile application allows agricultural officers to collect farmer related data. webapp along with cloud-based system allows officers to collect ad store almost all data related to farmers, farms, cultivation, harvest details, aid information (fuel, fertilizer, pesticides, monetary etc.),

crop disasters and disaster occurring f agriculture farms. The application has validation checks to ensure accurate and complete data entry.

**Farmer Data Entry**: Agricultural officers can enter detailed farmer profiles, including demographic information (name, age, region), land size, and cultivation methods.

- **Farm and Cultivation Details**: Officers can collect specific data on the types of crops grown, farm size, and location (using GPS data), irrigation methods, and expected yield.
- **Aid Distribution Records**: The system allows officers to record and track the distribution of various types of aid, including monetary support, fuel, fertilizer, and pesticides. The status of each distribution is updated in real time, ensuring transparency.
- **Disaster Reporting**: Officers can collect detailed information on agricultural disasters (e.g., droughts, floods), including the scale of the disaster, affected farms, and the type of damage sustained.
- **Data Validation**: Built-in validation checks ensure that required fields are completed, data formats are correct (e.g., date fields, numeric values), and duplicate records are minimized.

4. Data Representation

The system is designed to enhance user experience through interactive dashboards that feature customizable charts, graphs, and maps. It offers advanced filtering capabilities, allowing users to drill down into data based on specific criteria and parameters. Additionally, the system is equipped to generate visualizations that are compatible across a variety of devices and screen sizes, ensuring accessible and adaptable data representation.

- **Customizable Dashboards:** Different stakeholders (agricultural officers, regional managers, farmers) will have access to personalized dashboards tailored to their specific needs.
- **Interactive Visualizations:** The system supports the generation of dynamic charts, graphs, and geographical maps to visualize agricultural data trends (e.g., yield data, disaster impact, aid distribution).
- **Data Filters:** Advanced filtering options allow users to drill down into data by region, crop type, aid type, or disaster impact, making it easier to identify patterns or issues.

5. Messaging and Notifications

The web service allows broadcast messages to regional agricultural officers, farmers, and stakeholders when necessary.

- **Broadcast Messaging:** The system includes functionality to send urgent broadcast messages to all users (agricultural officers, farmers) in cases of emergency, such as natural disasters or sudden changes in aid policies.
- **Message History:** A log of sent messages is maintained for future reference.

6. Registering Users

The application enables the registration of farmers based on their regional location. Additionally, it allows for the registration of agricultural officers into the system. Individuals have the capability to self-register as generic users. Administrative personnel, including admins and agricultural field officers, are empowered to assign specific roles to the users who have registered, facilitating a structured and role-based access to the system's features.

**Self-Registration:** Generic users (e.g., farmers) can self-register via the web application by entering basic details such as name, location, and contact information.

- **Role-Based Registration:** Agricultural officers and admins will have elevated permissions to register new users within their jurisdiction or system. These officers can add and verify farmers, assigning them to specific regions.
- **User Verification:** A verification process will be required for certain users, such as agricultural officers, where admin approval is necessary to activate their account.
- **User Role Assignment:** Admins and field officers will be able to assign user roles (e.g., farmer, field officer, admin), with role-specific access and functionality.

7. Report Generating

Agricultural officers have the capability to generate necessary reports. Additionally, users have access to publicly available reports, allowing for broader dissemination and reference of important agricultural data and insights.

**Customizable Reports**: Agricultural officers can generate detailed reports based on the data collected, such as farmer profiles, cultivation progress, disaster impacts, and aid distribution.

- **Predefined Reports**: Standardized reports are available for common use cases, including monthly cultivation updates, disaster assessments, and resource distribution logs.
- **Dynamic Report Filters**: Officers can apply various filters (e.g., time period, region, aid type, crop type) to create customized reports, aiding in targeted decision-making.
- **Publicly Accessible Reports**: Certain reports (e.g., general agricultural trends or disaster impact summaries) will be made available to the public for broader transparency and insight-sharing.
- **Export Functionality**: Reports can be exported for offline analysis or presentation.

### 2.4.4 Non - Functional Requirements

1. Performance

The system should have fast response times for efficient data entry and retrieval. The system should handle a large volume of data without performance degradation. The cloud-based system should provide quick and reliable access to stored agricultural data.

2. Reliability

The system should be reliable and stable, minimizing crashes and errors. The system should have backup and recovery mechanisms to prevent data loss. The cloud-based system should ensure high availability and minimal downtime.

3. Usability and User Experience

The software should have an intuitive user interface, providing a seamless and user-friendly experience. It should be accessible across different devices and platforms, ensuring a consistent user experience. It should be accessible to users with varying levels of technological proficiency

4.Scalability

The software should be capable of handling a large volume of concurrent users and data processing tasks without compromising performance. It should scale effectively to accommodate increasing user demands.

5. Security

The application should employ robust security measures to protect sensitive agricultural data. It should ensure data confidentiality, integrity, and availability through secure data transmission protocols and authentication mechanisms.

6.Integration

The mobile application and cloud-based system are designed to integrate flawlessly, facilitating uninterrupted data synchronization and sharing. This integration may extend to include third-party services, aiming to boost the system's functionality and improve data exchange processes. For further details on this integration, including specific methodologies and technical guidelines, Chapter 4 offers comprehensive insights.

## 2.5 Proposed Development Process Model

The iterative waterfall model is chosen for to adopt in the project; practical mobile app development and web system development. Because, it provides clear project phases, accommodates stable requirements, involves stakeholders, ensures integration and testing, facilitates early risk mitigation, and enhances efficiency (Bhatnagar, 2015). In practical software development, the iterative waterfall model addresses this by incorporating feedback paths for error detection and correction within the same phase, rather than waiting until the project's end. This model combines sequential steps with iterative design, allowing improvements and changes at each stage. Iterative waterfall model includes stages such as requirements gathering, design, implementation, testing, and deployment. This approach ensures flexibility and progress throughout the development process. Figure 2.2 showcases the Iterative Waterfall model i.e. a popular and traditional approach in practical software development.

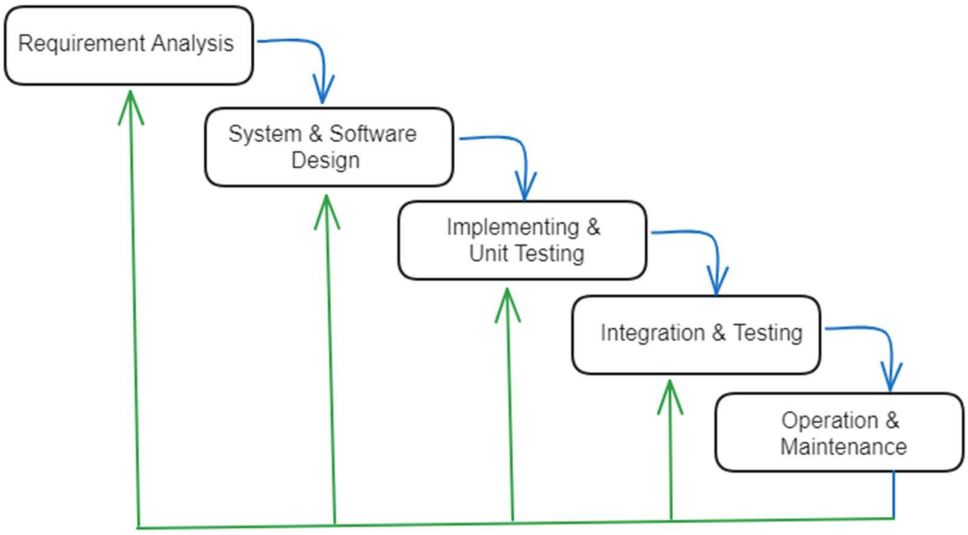



*Figure 2.2: Iterative Waterfall Model (Bhatnagar, 2015)*

## 2.6 Summary

Chapter 2 provides the background information for the proposed system aimed at enhancing the agriculture sector in Sri Lanka. It highlights the importance of agriculture for the country's economy, job creation, and food security. The chapter emphasizes the challenges faced by the agriculture sector due to limited access to timely and accurate data, which hampers decision-making and resource allocation. The chapter further explores the related technologies used in current agricultural software systems.

The chapter reviews similar systems and technologies that have been utilized in the agriculture industry in Sri Lanka and globally. It mentions cloud-based agricultural management software systems, mobile applications, and sensor technologies that have been used to optimize and streamline farming practices.

The chapter discusses the existing system processes and functionalities, focusing on 'Agrimanager' (Agrimanager) as a closely aligned system with similar requirements and highlights the features and capabilities of it that can be incorporated and enhanced in the proposed system.

The chapter concludes with a requirement analysis, which emphasizes the need for a comprehensive cloud-based system for data acquisition, representation, and informed decision-making in the agriculture sector. It outlines the scope of the proposed system, including mobile data collection, cloud-based data storage, market connectivity, communication, and data analysis. The user classes and characteristics are also identified, including agricultural officers, farmers, administrators, and researchers.

# Chapter 3 – Design Architecture

## 3.1 Introduction

The agricultural sector plays a pivotal role in the livelihoods of the Sri Lankan population but is currently impeded by significant decision-making challenges due to the lack of comprehensive, island-wide agricultural data. This project aims to address these challenges by leveraging technology to create a centralized agricultural database. By utilizing cloud technology, along with mobile and web applications, the initiative seeks to modernize traditional agricultural practices, thereby enhancing decision-making efficiency and boosting productivity in farming. This technological integration promises to transform the agricultural landscape in Sri Lanka, making it more data-driven and efficient.

The project design comprises several key components: a mobile application, a webapp, a cloud-based platform, and databases. Together, these elements facilitate data collection, accessibility, and decision-making support for users. The mobile app empowers regional officers to gather agricultural data directly from farmers and update the system's database. The webpp serves as a management tool for agricultural data and facilitates report generation. Additionally, a broadcast message service enhances efficiency in the agricultural sector by ensuring timely and reliable dissemination of information. All data is securely stored in a cloud database, enabling informed decision-making through data representation. Reports aid users in visualizing and strategizing based on agricultural data. The project also offers free advertising services for farmers, enabling direct communication with buyers and government oversight of pricing and regulation. This ensures fairness and informed decision-making regarding price changes. Furthermore, researchers can leverage the centralized agriculture data to support informed decision-making and contribute to the improvement of Sri Lanka's agricultural sector. Overall, the project optimizes data utilization and fosters collaboration for the betterment of Sri Lankan agriculture.

This chapter presents visual representations and it collectively provides a comprehensive perspective of the system’s design. The design elements harmonize user requirements, system functionality, and data management.

## 3.2 Design Strategies Used

Design strategies are methods that help to arrange sections of a program in a way that's simple to create and modify. In this project, Object Oriented Design (Hillar, 2015) is used and, objects, classes, encapsulation, inheritance, polymorphism are some major features which were adopted. This means we focus on the things in the system (entities) and what makes them unique, rather than just the tasks the software does. Object Oriented Design makes it easier to manage complexity and create efficient and reusable implementation.

The top-down design approach (Münch, 2022), is used because it helps to understand and build the system step by step. It highlights that systems consist of multiple sub-systems and components, forming a hierarchical structure (Münch, 2022). Top-down design involves initially considering the entire software system as one entity and breaking it down into sub-systems or components based on certain characteristics. Each of these is then treated as its own system and further broken down. This process continues until the lowest level of the system hierarchy is reached. Starting from a generalized model, top-down design progressively defines more specific parts of the system, culminating in the complete system once all components are integrated (Münch, 2022). Multitier architecture is adopted in this project as it promotes separation of concerns, modularity, and ease of maintenance. Each layer communicates with adjacent layers through well-defined interfaces, and changes in one layer have minimal impact on other layers.

Modularization involves breaking down a software system into smaller, self-contained modules. Each module focuses on a specific aspect of functionality. Figure 3.1 illustrates Modules in “Ceylon AgriData” System and it contains its own set of functions, logic. This makes it easier to work on individual modules independently and allows for better code organization, enhances reusability, minimize risks, error conflicts etc.

"Ceylon AgriData" comprises two primary components: a mobile application and a web app. The following sections will provide detailed explanations of each component.

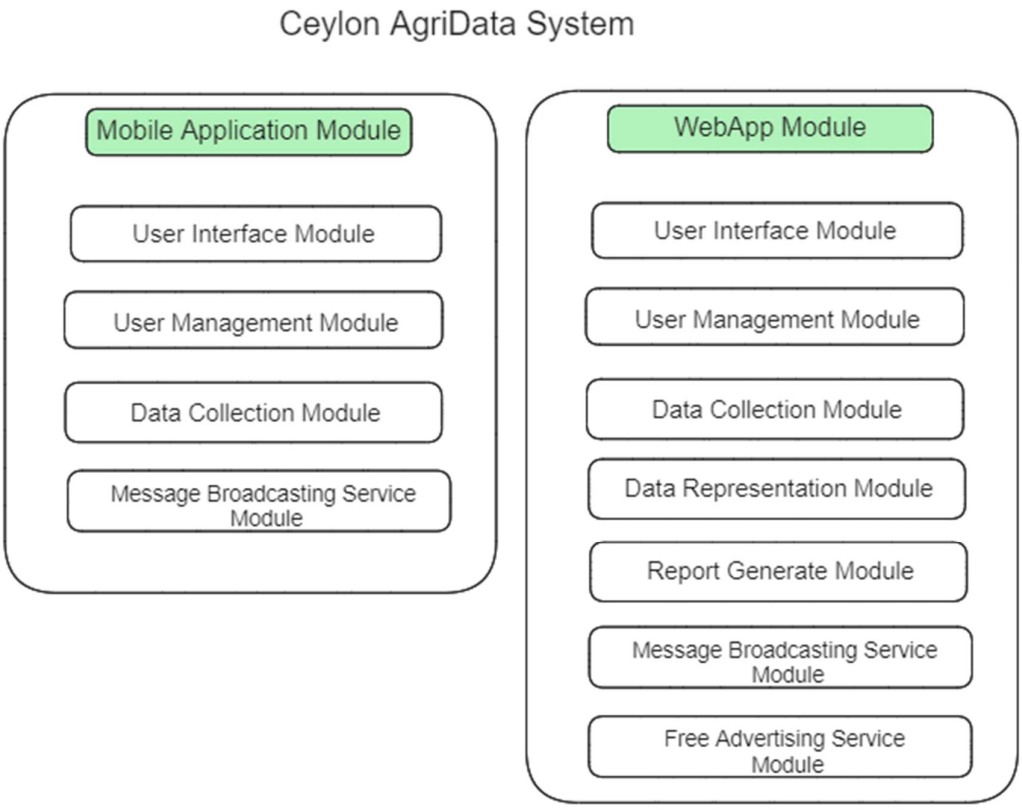



*Figure 3.1: High-level representation of modules in "Ceylon AgriData" System*

Table 3.1 outlines the design modules for the mobile application component, while Table 3.2 provides a summary of the design modules for the web application component within the "Ceylon AgriData" system.

*Table 3.1: Modules designed for mobile application designing*

| Module Name | Description |
|---|---|
| User interface module (UI) | All UI s of mobile application |
| User management module | User management related functionalities of users (basically CRUD operations of farmers) |
| Data Collection module | Data collection related functionalities (data on agricultural entities such as farms, |

| | |
|---|---|
| | cultivation, aids like fertilizer, pesticides, fuel, and information of farmers etc.) |
| Message Broadcasting Service module | Broadcasting messages to farmers by agricultural field officers |

Figures 3.2 – 3.17 represents the designed UI s of mobile application.

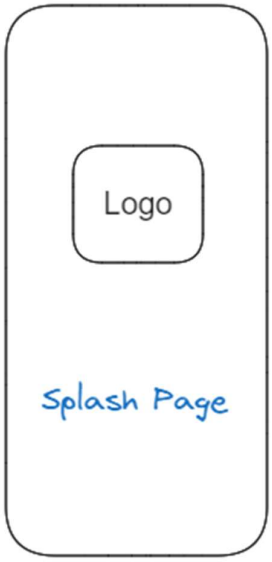



*Figure 3.2: Splash page designing of mobile application*

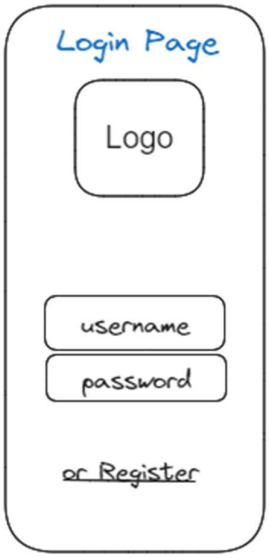



*Figure 3.3: Login page designing of mobile application*

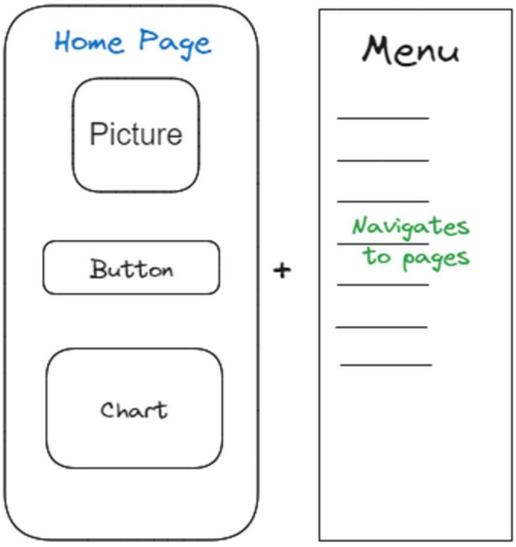



*Figure 3.4: Home page designing of mobile application*

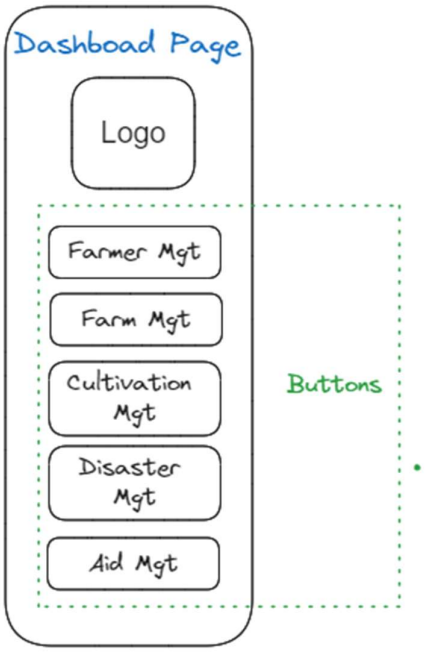



*Figure 3.5: Dashboard page designing of mobile application*

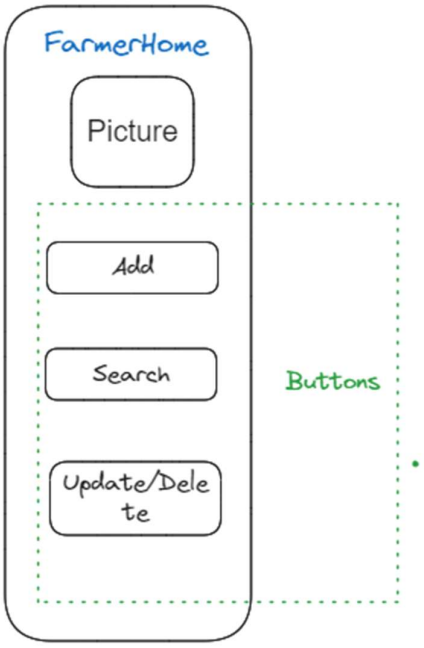



*Figure 3.6: Farmer manager home page designing of mobile application*

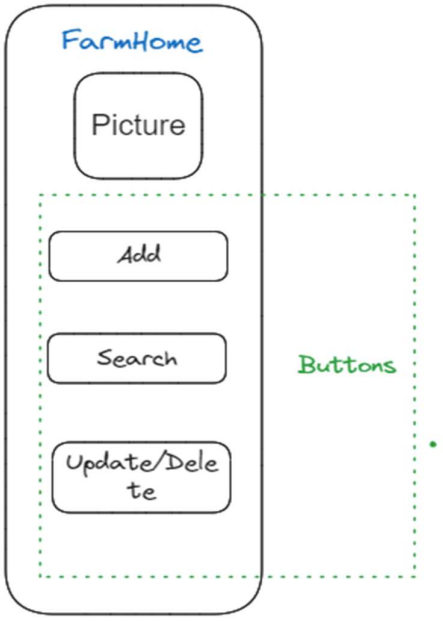



*Figure 3.7: Farm manager home page designing of mobile application*

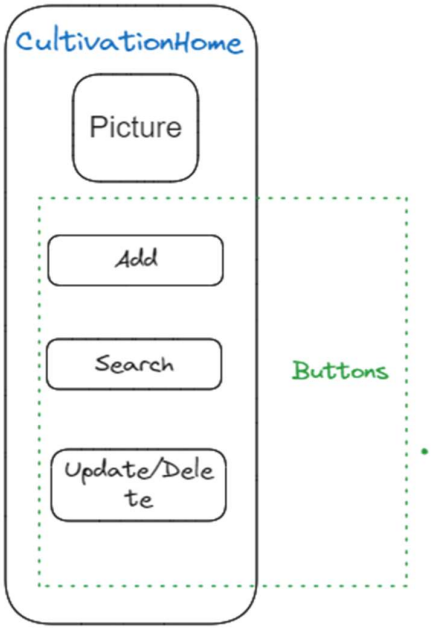



*Figure 3.8: Cultivation manager home page designing of mobile application*

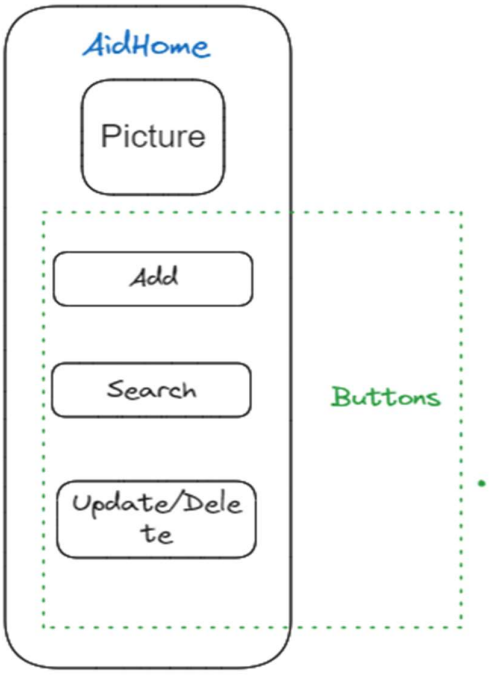



*Figure 3.9 : Aid manager home page designing of mobile application*


*Figure 3.10: Register farmer page designing of mobile application*

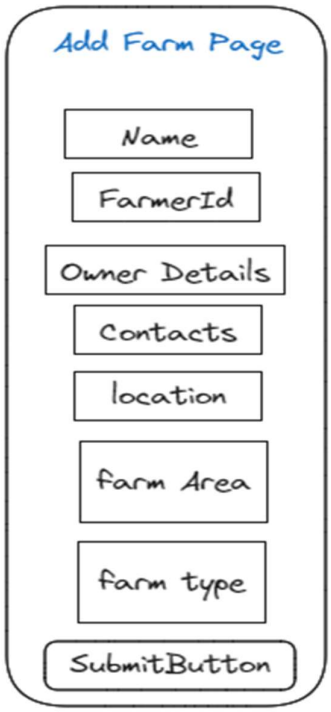



*Figure 3.11: Add farm page designing of mobile application*

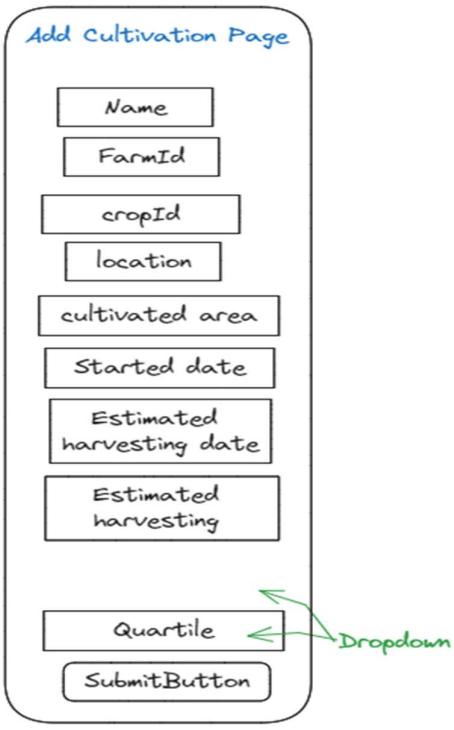



*Figure 3.12: Add cultivation page designing of mobile application*

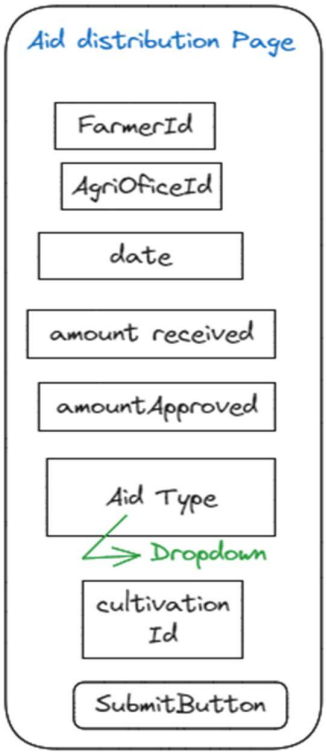



*Figure 3.13: Add cultivation page designing of mobile application*

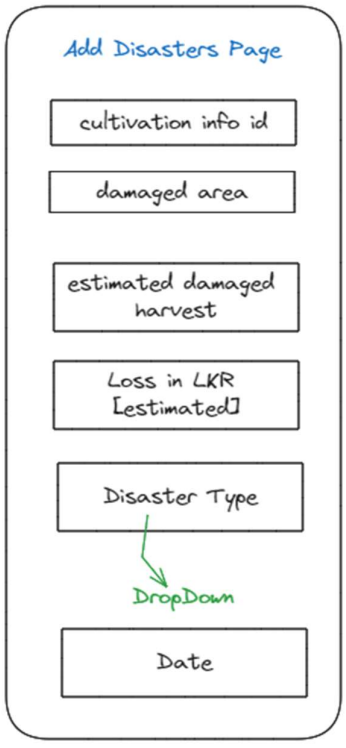



*Figure 3.14: Add disaster page designing of mobile application*

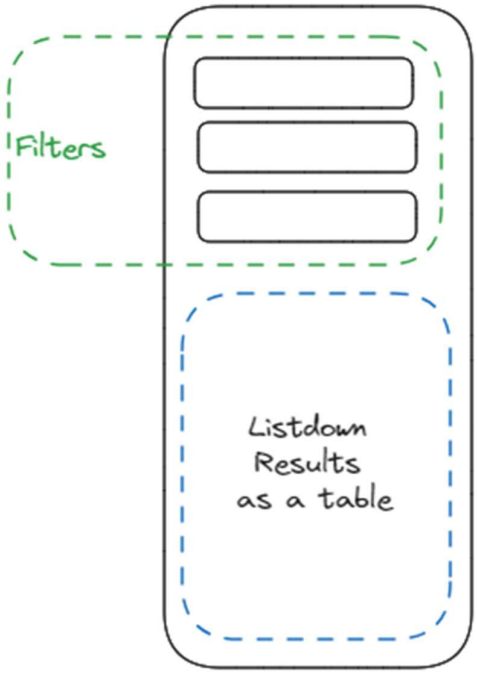



*Figure 3.15: Search operation page of designing of mobile application*


*Figure 3.16: Page for update / delete operations of designing of mobile application*

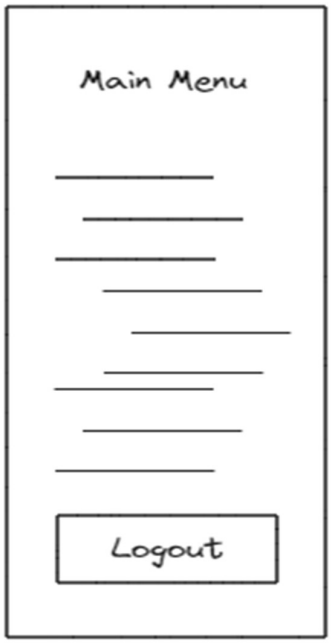



*Figure 3.17: Logout button at main menu in designing of mobile application*

*Table 3.2: Modules designed for web application designing*

| Module Name | Description |
|---|---|
| User interface module (UI) | All UI s of web application |
| User management module | User management related functionalities of users (CRUD operations of users: farmers, agriculture officers, researchers etc.) |
| Data collection module | Data collection related functionalities (data on agricultural such as farms, crops, cultivation, aids like fertilizer, pesticides, fuel, monetary and information of farmers, researchers etc.) |
| Data representation module | Representation of collected agriculture data in a comprehensive manner using tables, graphics such as pie-charts, bar charts and maps. |

| | |
|---|---|
| Report generate module | Use filters to make reports to facilitate data driven decision making. Filtered data can be downloaded as .csv files. |
| Message broadcasting service module | Broadcasting messages to farmers, officials by agricultural field officers and admin. |
| Free advertising service module | Functions related to advertising advertisements of registered farmers. |

Figures 3.18 to 3.21 depict the designed user interfaces (UIs) of the web application.

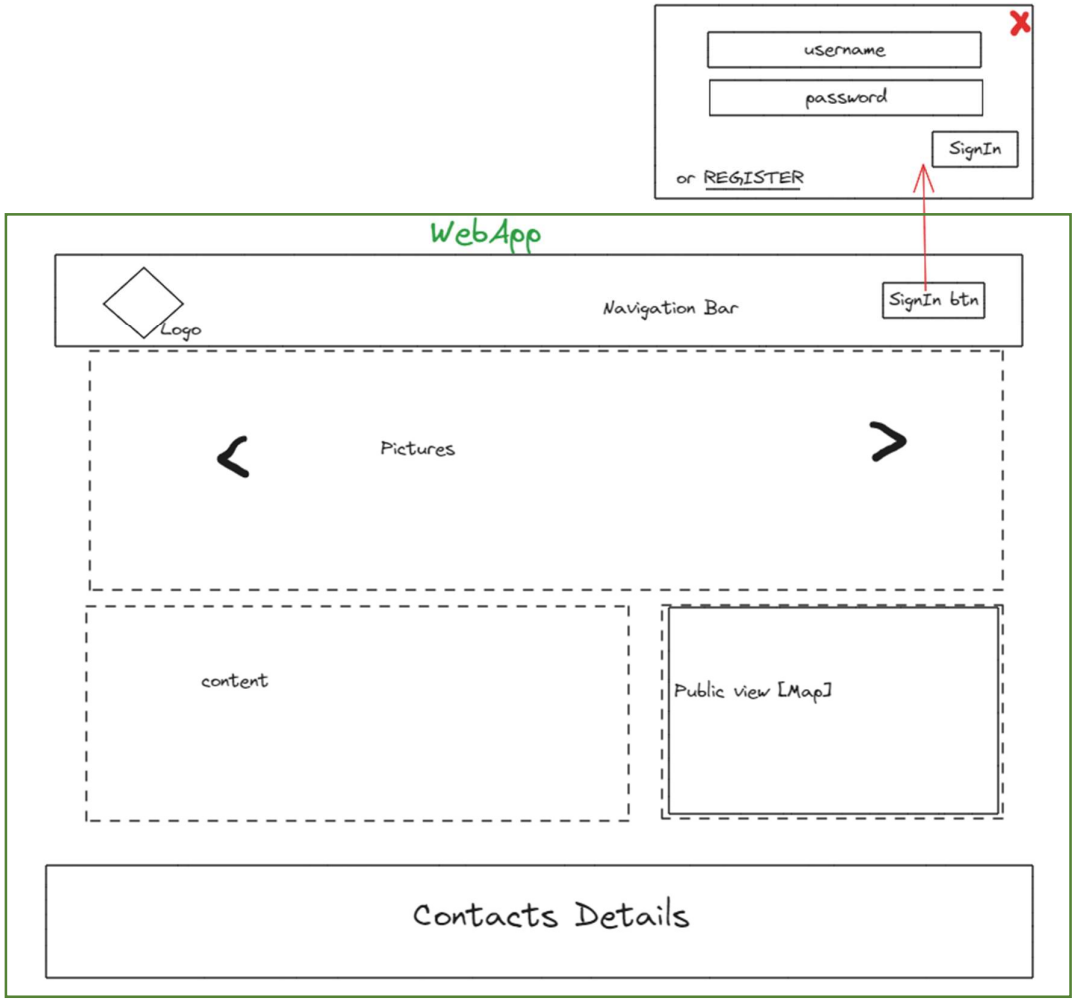



*Figure 3.18: designing of UI - landing page of react webapp*

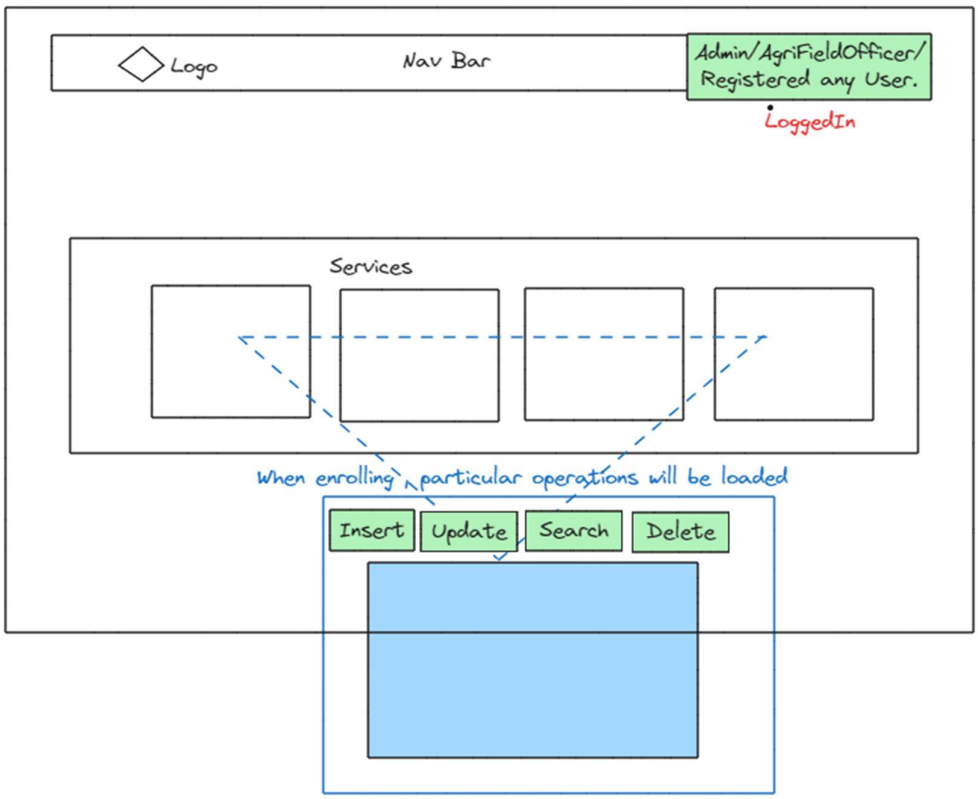



*Figure 3.19: Designing of UI of webapp for data managing related operations*

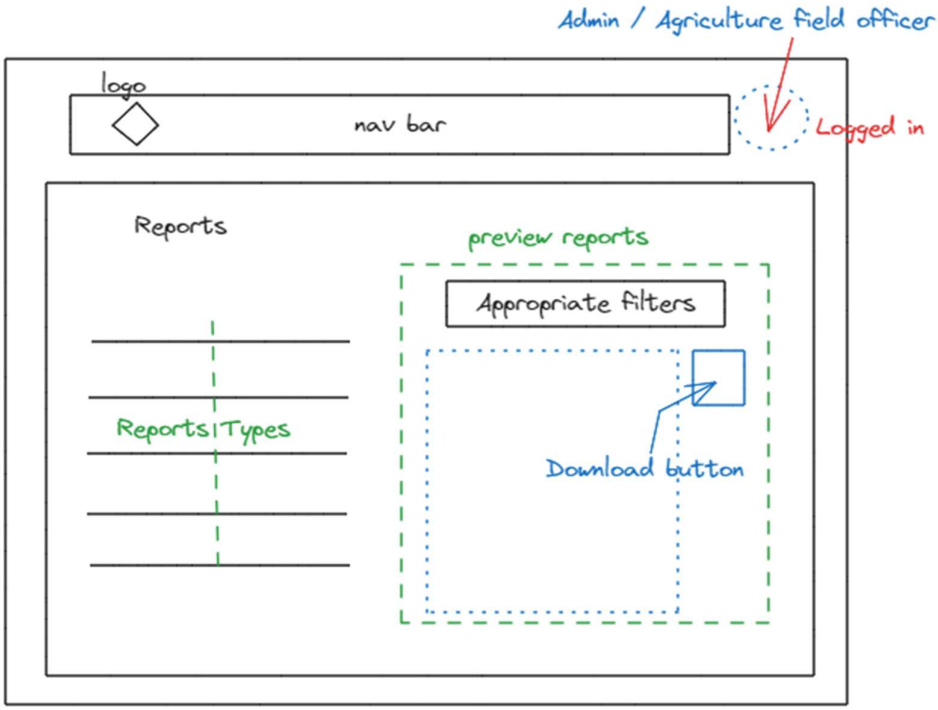



*Figure 3.20: Report generating page design of webapp*

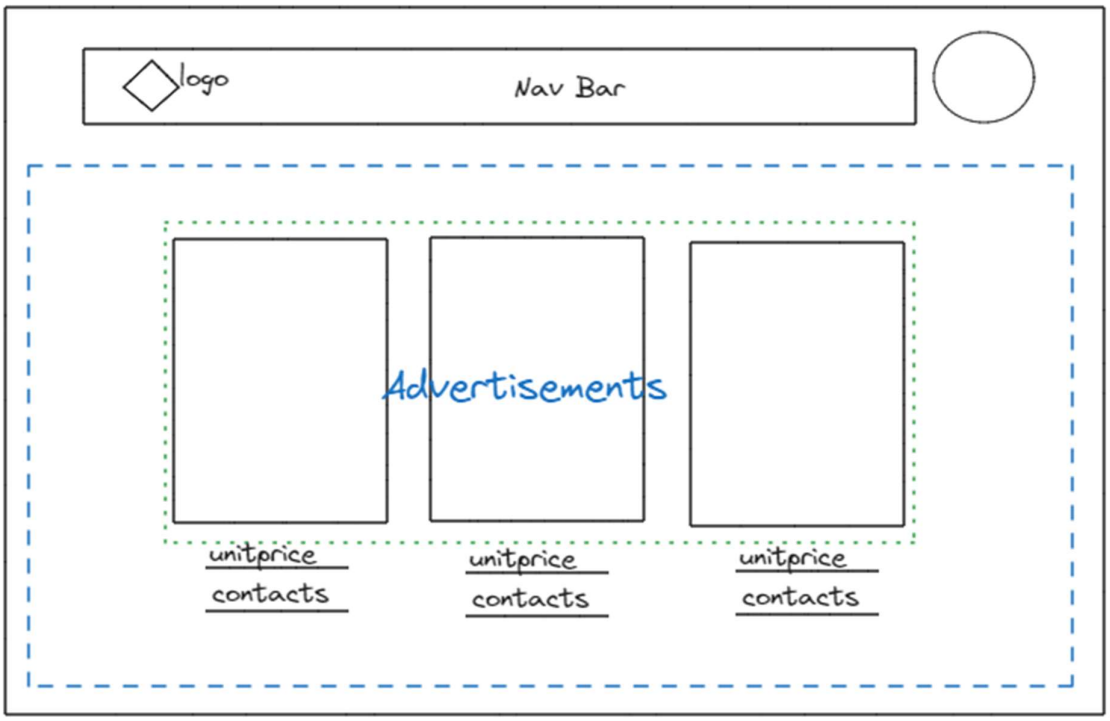



*Figure 3.21: Advertisement service page designing of webapp*

## 3.3 System Architecture

The layered architecture pattern arranges components into horizontal layers, each with a distinct role in the application. While the number and types of layers can vary, a common setup includes four layers: presentation, business, persistence, and database, which can be combined based on needs, resulting in three to more than five layers in different applications (Richards, 2022).

In the project, service-oriented multi-tier architecture is employed, featuring distinct layers: presentation (mobile and web applications), application logic (REST web service), and data storage (database), ensuring effective separation of roles. This approach promotes scalability and maintainability, as alterations in one layer and, do not impact the others, with REST APIs facilitating communication between the presentation and application logic layers (Petrillo et al., 2016).

The presentation layer handles user interaction and interface, the application logic layer manages processing and business rules, and the data storage layer handles data management

and storage. This separation of concerns enhances scalability, maintainability, and reusability by isolating different functionalities into well-defined layers (Richards, 2022).

Figure 3.22 depicts the high-level solution architecture of the system. Presentation layer exposed interfaces for system consumers such that agriculture officers (admins, agriculture field officers), Farmers, Researchers, generic users etc. Application logic layer includes the API service and the web app which handles all the business processes and logic in the considered domain such as agricultural data management (insert, update, delete agricultural data, search options, authorization, login/logout functionality, report generating functions, message broadcasting functionality, free advertising service functionality etc.). Data Layer holds all the domain related data.

Python language is used to build the backend of the web application as it supports for building scalable and maintainable web applications using different web frameworks/libraries like flask, Django etc. Python is renowned for its high-level nature, readability, and extensive library of modules and packages, which contribute to faster and more efficient development processes. Flask is used as the web framework for building the application as it is flexible and easy to use, making it a popular choice for building RESTful APIs. A relational database - MYSQL database is used for the data storage requirement of the application. React is used to build Web application front end. The mobile application is built using flutter for android. As the application was built as a modular service, like react based web frontend application, python-based flask API, and data persistence, any supportive cloud service can be used to deploy this system in production convincing that the design of the project more flexible.

In addition, cloud-based systems, integrating APIs, web applications, and cloud-hosted databases, yield scalable, cost-efficient solutions with global accessibility, fostering rapid development and high reliability (Petrillo et al., 2016). This approach enables seamless integration, ensures security and compliance, and streamlines maintenance, empowering enterprises to innovate effectively and maintain competitiveness.

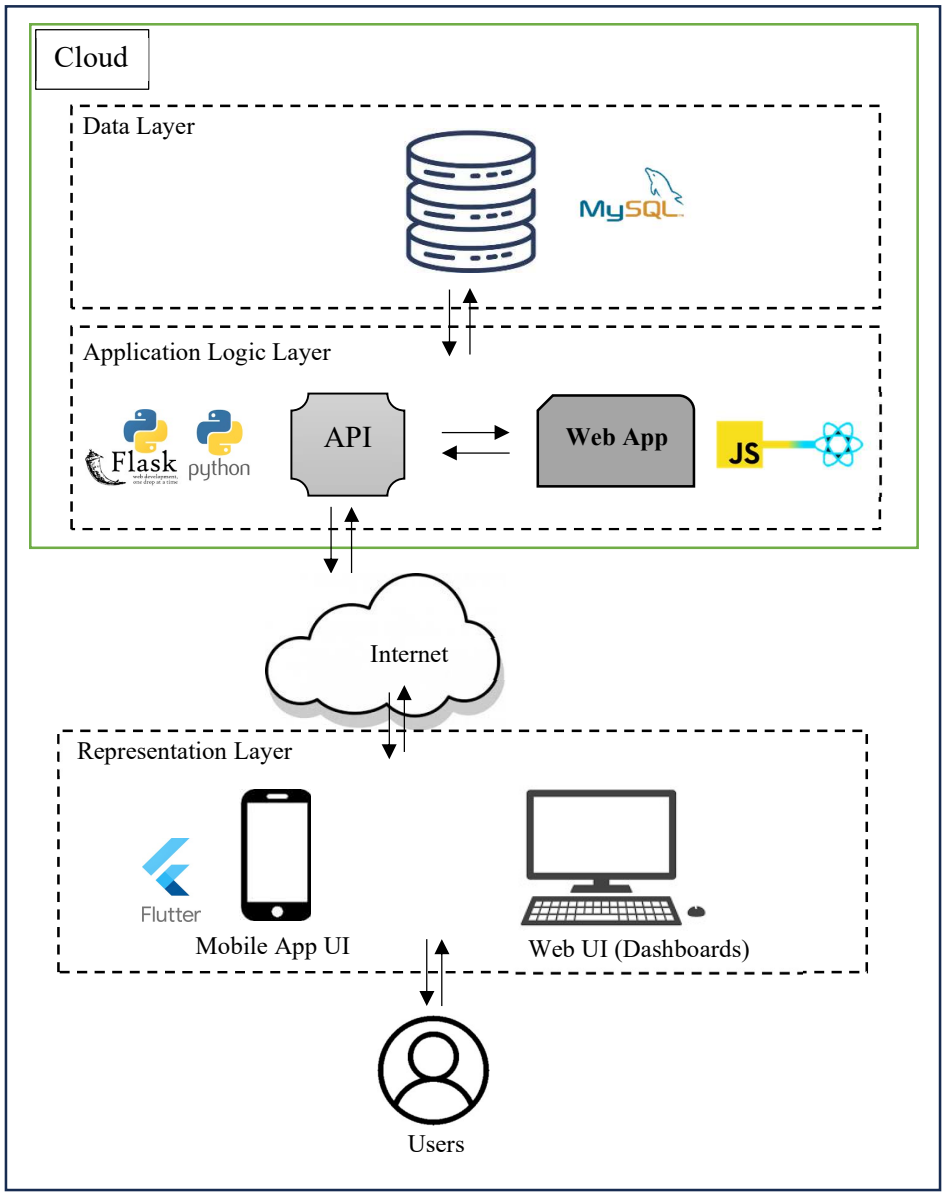



*Figure 3.22: High-level solution architecture of the "Ceylon AgriData" system*

### 3.3.1 Frontend and Backend Module Overview within the System Architecture

The design of the “Ceylon AgriData” system is structured around two core separate services: the frontend and the backend. The frontend serves as the GUI, encompassing both the mobile and web application UIs, while the backend handles business logic, data persistence, and communication scenarios as indicated in Figure 3.23. Integration with third-party services like Gmail API and mapping tools enhances functionality.

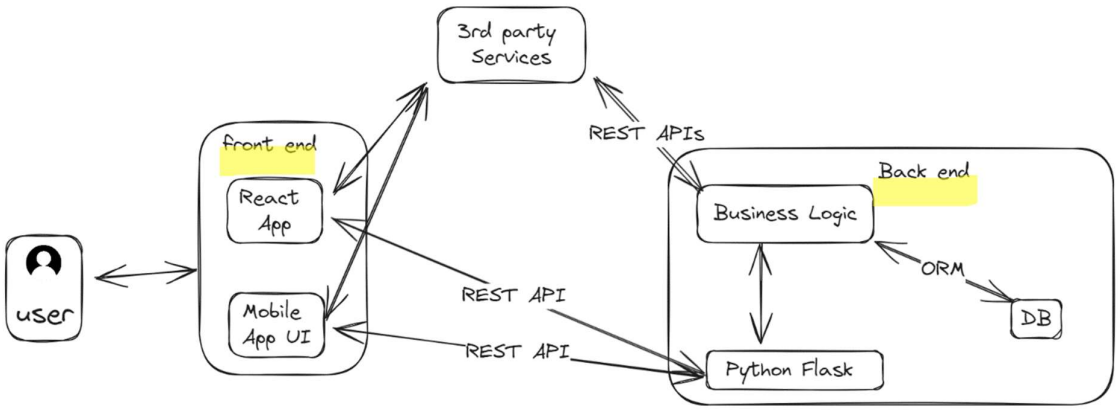



*Figure 3.23: Frontend and Backend overview of the "Ceylon AgriData" system*

In the frontend, the API service manages all HTTP requests centrally, using the Axios library for efficiency. The React Components module maintains UI elements for a unified user experience, ensuring clarity and modularity in development. The backend comprises key modules including 'Models', 'Schemas', and 'Routes'. Models define entity structure, while Schemas provide structured data representation for consistency. Routes coordinate data flow and operations, exposing endpoints for client interaction. High level diagram of modules designed as per system architecture is in Figure 3.24.

Together, these modules form a robust system for agricultural management, facilitating efficient data handling and functionality implementation. This structured approach ensures coherence, scalability, and seamless communication between frontend and backend components.

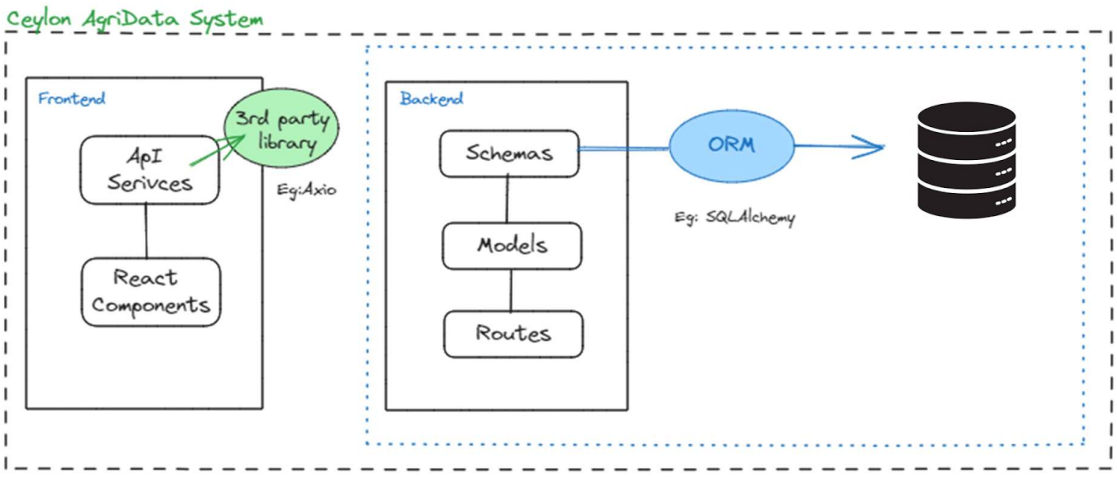



*Figure 3.24: High level diagram of modules designed as per system architecture*

## 3.4 Justification of selecting React and Flask in the project

React and Flask were chosen for their strengths: React's component-based architecture simplifies UI management, virtual DOM optimization ensures faster rendering, and state management is enhanced with React Hooks. Flask's lightweight framework and support for various ORMs streamline development. Both benefit from rich ecosystems of third-party libraries for additional functionality, as outlined in Table 3.3.

*Table 3.3: Justification of selecting React and Flask in the project*

| React | Flask |
|---|---|
| **Component-based architecture**<br>React allows to build UIs using component-based architecture, making it easier to manage and reuse. | **Framework**<br>Flask uses Python, a dynamically typed language. No need to declare types.<br>Flask is a lightweight web framework for Python. |
| **Virtual DOM**<br>React's virtual DOM implementation optimizes performance by only updating the parts of the DOM that have changed, resulting faster rendering | **Project Structure**<br>Flask projects typically have a simpler structure |
| **State Management**<br>React provides various options for managing application state, such as React's built-in state management; React Hook functions such as useState and useEffect. | **Configuration**<br>Flask uses Python code for configuration, typically in the form of function decorators or configuration files. |
| **Third-party libraries**<br>Axios – HTTP client for the browser and handles responses in react applications,<br>React-Leaflet – Integrate maps in react applications,<br>React-Bootstrap – pre-designed UI components for rapid development etc. | **ORM**<br>Flask supports various ORMs (Object-Relational Mappers) like SQLAlchemy for interacting with databases. |

## 3.5 UML Diagrams

Unified Modeling Language (UML) diagrams are essential tools in software development, offering standardized visual representations for design, structure, and behavior aspects of a system and these diagrams bring multiple advantages, facilitating comprehension, communication, and implementation of intricate software systems (Dennis, 2012).

### 3.5.1 Use case analysis

Use cases are used to describe how users interact with a system to achieve tasks. They help to understand the steps needed for users' goals and can lead to detailed functional requirements stated in chapter 2. Use cases are important for various development methods and are especially helpful for systems involving user interactions. Figure 3.25 represents the use case diagram of the system, “Ceylon AgriData”.

Actors of the system are the persons who interact with the system;

- Admin : Person who has access to overall system “Ceylon AgriData”.
- Agricultural field officers: Person who has access only to privileged area.
- Farmer, researcher, generic user are the persons who interact with the system to have services.

Use case descriptions of particular actors are as follows.

1. **Admin:**

- Admin logins/sign in the system
- Manage agricultural field officers and other users
  - Admin verify other users sign-up requests.
  - Assigns officers to specific fields as needed.
- Manage aid funds information
  -Manages funding aids receives (fuel, fertilizer, pesticides, monetary etc.) by the government in the system
- Generate Reports
  -Generates various reports including officer details, crop cultivation, farmer details, aid distribution, etc.
- Broadcast messages
  -Broadcasts messages to officers for reliable message dissemination
- View public reports

- View published advertisements on webapp
- Sending requested data to researchers and maintain its records

**2. Agriculture Field Officer:**

- Registration

  -The agriculture field officer signs up via the mobile application or webapp, providing necessary details.

  - A request is sent to the regional admin for verification and approval.

  - Upon approval, the agriculture field officer receives an email confirmation and gains access to the system.

- Manage Farmers

  - After logging in, the Agriculture Officer adds/updates/views farmer details through mobile or web applications. (Using mobile application in field visits are more convenient)

- Manage Farm Information

  - logs in the system and adds/updates/views farm details via the mobile or web application.

  - Geographical location can be added via the mobile application during field visits.

  - or, manually can be added using web application.

- Manage Cultivation information

  - Add/update/view cultivation details and upload them to the system.

  - Cultivation details can be updated at any time, and past records are accessible.

  - Eligible farmers must be registered in the system beforehand.

- Manage Fertilizer Distribution

  - Update the system with aid distribution details for their assigned area.

  - Eligible farmers must be registered in the system beforehand.

- Manage Fuel Distribution

  - Updates the system with aid distribution details for their assigned area.

  - Eligible farmers must be registered in the system beforehand.

- Manage Pesticides Distribution

  - Update the system with aid distribution details for their assigned area.

  - Eligible farmers must be registered in the system beforehand.

- Manage Monetary Distribution

-Update the system with aid distribution details for their assigned area.

- Eligible farmers must be registered in the system beforehand.

- Manage miscellaneous aid distribution

  - Update the system with miscellaneous aid distribution details for their assigned area.

  - Eligible farmers must be registered in the system beforehand.

- Manage Disaster Information

  - Update disaster damage details for affected cultivations in their field area.

- Manage Disaster Information

  - Update disaster damage details for affected cultivations in their field area.

- Verify Advertisements of farmers in free advertising service portal

- Generate Reports

  -Generate reports on cultivation, farmer, disaster damage, and aid distribution details with various filters.

- View public reports on web application

- View published advertisements on webapp

- Broadcast messages to farmers

**3. Farmer:**

- Registration

  - Farmers are registered to a field area by the Agriculture Officer.

  - Upon registration, farmers create an account in the system for marketplace access.

- Publish advertisements

  - Registered farmers publish/view/update/remove advertisements for their harvest in the marketplace.

  - Advertisements are verified by the agricultural field officer before publishing.

- View public reports and dashboard

  - Farmers generate reports on cultivation and harvest information.

- View published advertisements on webapp

**4. Researcher**

- View Public Reports/Dashboard

  - Researchers view public reports and dashboard on agricultural data.

- View published advertisements on webapp

- Request data

  - Researchers contact agriculture office by sending messages through "contact us" feature to request data.

5. **Generic User**

- View Public Dashboard and Reports

  - Users view public dashboard and reports provided by the system.

- View published advertisements on webapp
- Request data

  -Contact agriculture office by sending messages through "contact us" feature to request data.

Admins and agriculture field officers have the capability to generate reports essential for official tasks and informed decision-making within the sector. The system provides various data representations and supports the download of agriculture data to facilitate this process. Users can specifically generate reports using filters available in the report section. *Appendix A* provides a comprehensive overview of the MIS reports that can be generated and visualized within the "Ceylon AgriData" system.

### 3.5.2 Process Modelling

Process modeling is a technique used to visually represent and describe the sequence of activities, tasks, and interactions within a system or a workflow. Process modeling helps in understanding the flow of work, identifying potential bottlenecks or inefficiencies, and improving the overall efficiency and effectiveness of a process (Dennis, 2012).

Manage users (registration, authenticate login),, manage Agri-Information (farms, cultivation, disasters etc.), manage Agri-aid information (money, pesticides, fertilizer, fuel etc.), manage free advertising service, data processing (upload and retrieve to cloud), data representation (represent Agri-Information for decision making etc.), broadcasting message service are included the main business processes in the "Ceylon AgriData" System.

Activity diagrams are the commonly used graphical representations to illustrate flow of activities, actions and decisions in the system. Figure 3.26 represents the activity diagram for registering a new farmer and add his farm and cultivation details using mobile application by agriculture field officer.

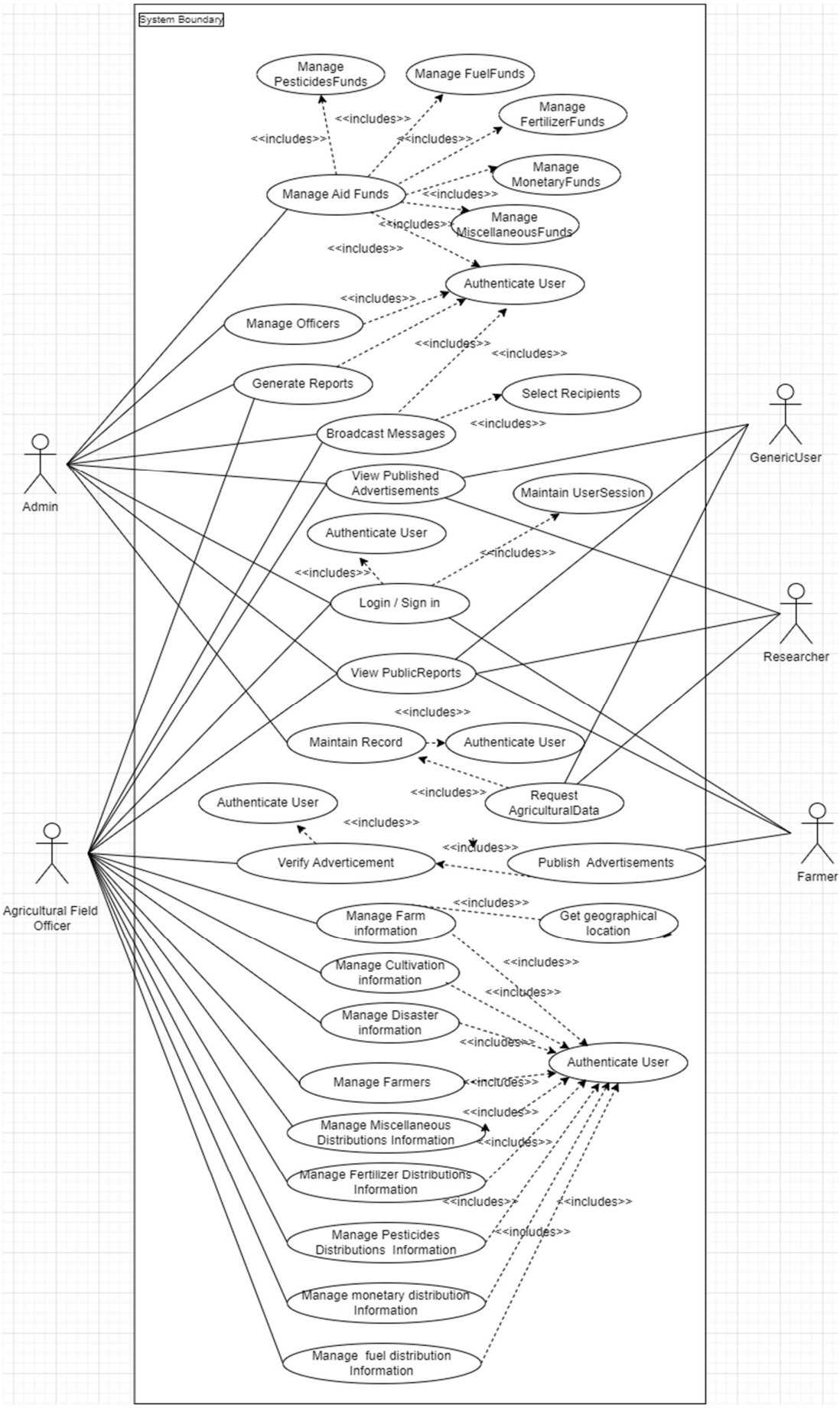



*Figure 3.25: Use case Diagram of the System "Ceylon AgriData"*

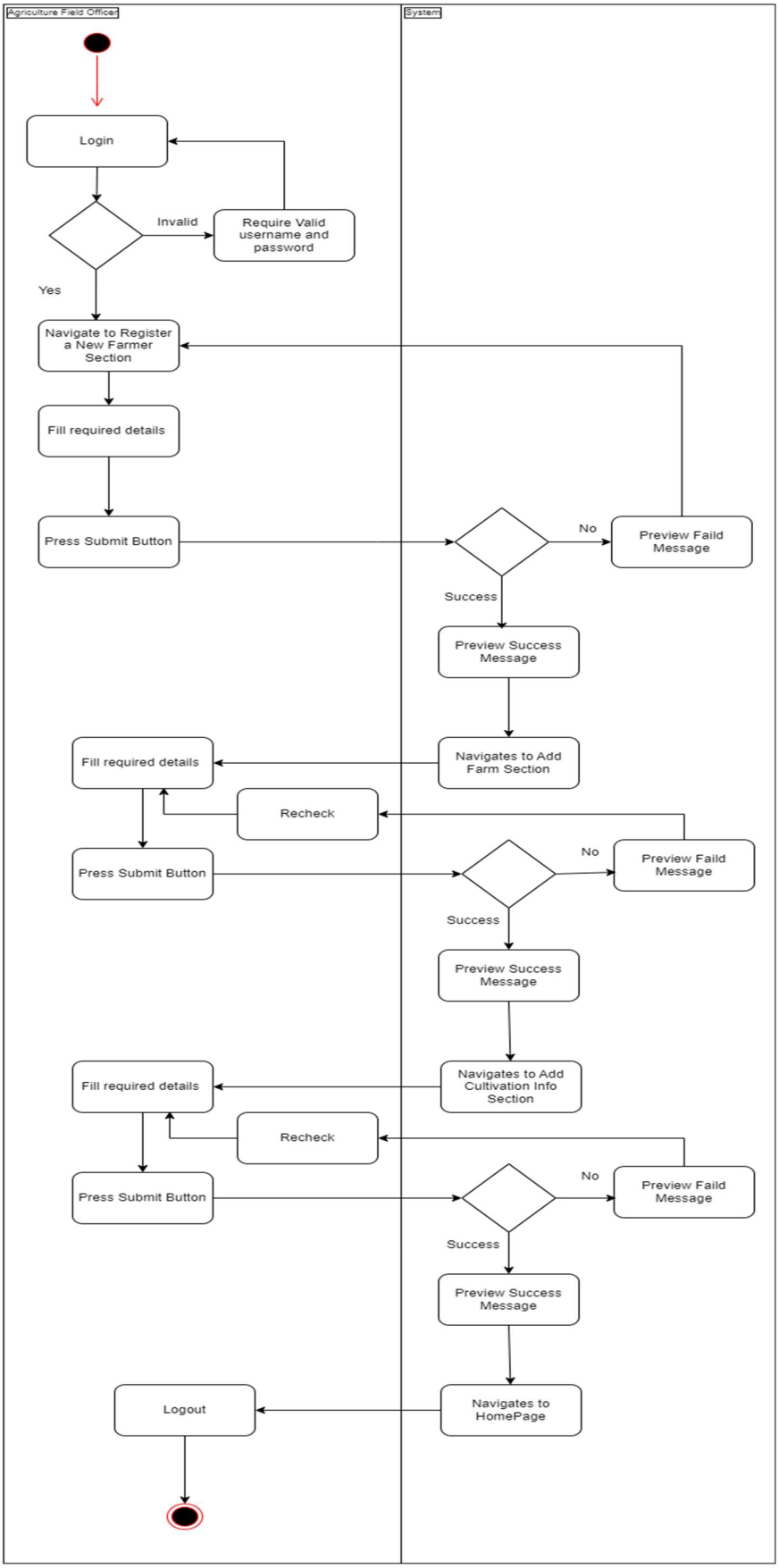



*Figure 3.26: Activity diagram for register a new farmer and adding his farm, cultivation details into the system by Agri field officer*

Figure 3.27 depicts the activity diagram illustrating the process of generating agriculture reports by an agriculture field officer. And, Figure 3.28 illustrates the activity diagram outlining the procedure for advertising reports by a farmer.

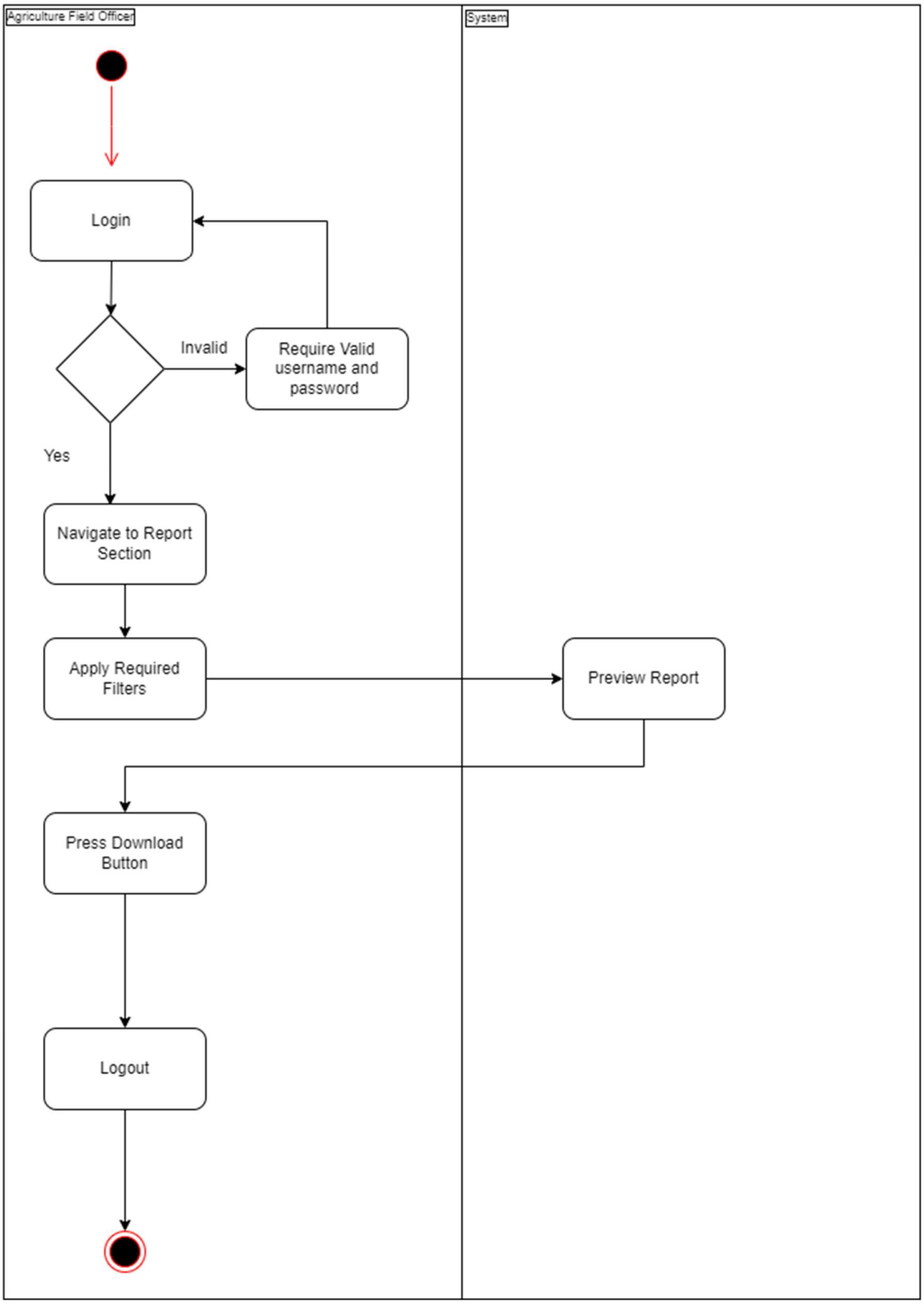



*Figure 3.27: Activity diagram to generate agricultural reports by a agriculture field officer*

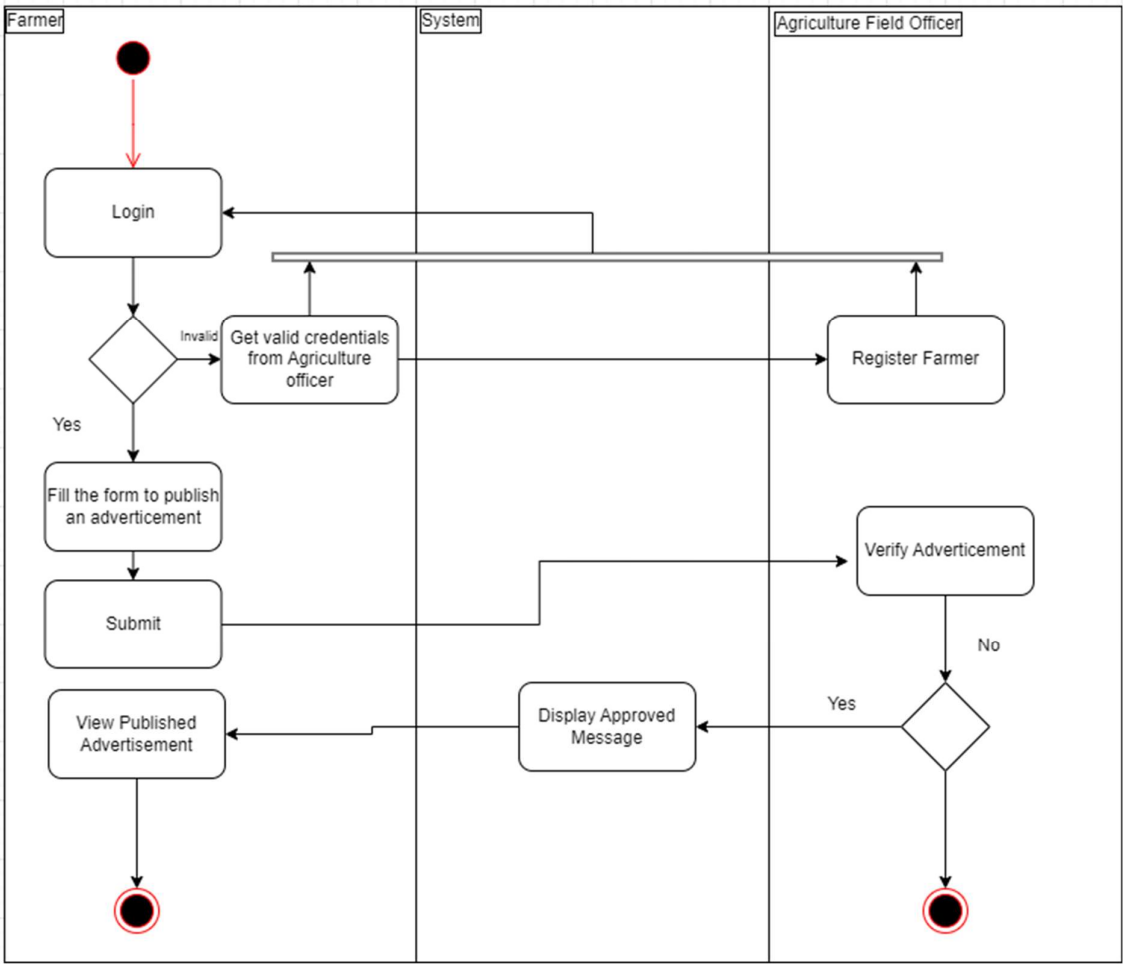



*Figure 3.28: Activity diagram for publishing advertisements by farmer*

### 3.5.3 Data Modelling

Data modeling in software design involves creating a structured representation of how data will be organized, stored, and interacted with within a software system. It assists in designing a database schema and ensuring that data is accurately captured and processed according to the software's requirements. All data related to various business processes is stored in the data layer. For a detailed description of the tables used in the database pertaining to these data layer functionalities, please refer to *Appendix B* and *Appendix C*. Figure 3.29 illustrates the database model of the “Ceylon AgriData” system.

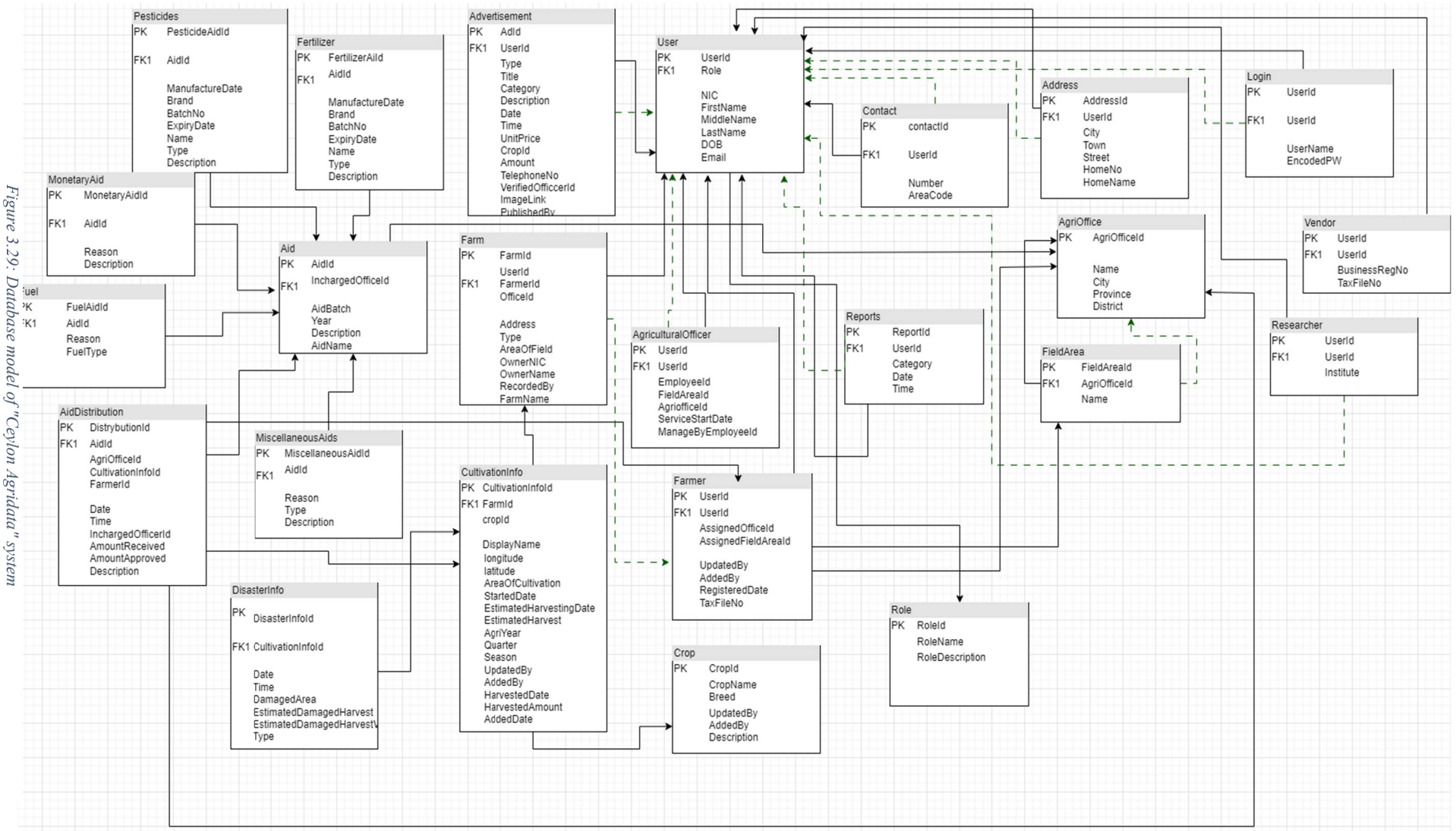



*Figure 3.29: Database model of "Ceylon Agridata" system*

## 3.6 Summary

Chapter 3 presents the comprehensive design architecture for the "Ceylon AgriData" system, aimed at addressing decision-making challenges in Sri Lanka's agricultural sector through technology integration. The design encompasses a mobile application, a website, a cloud-based platform, and databases, facilitating data collection, accessibility, and decision-making support. Utilizing Object-Oriented Design and a top-down approach, the system achieves modularity and scalability. The architecture follows a service-oriented multi-tier approach, with distinct layers for presentation, application logic, and data storage, ensuring effective separation of concerns and promoting scalability and maintainability. Leveraging Python, Flask, React, and MySQL, the system boasts a flexible and efficient backend, complemented by a robust frontend. The chapter provides a detailed breakdown of system modules, frontend-backend interactions, design strategies, justification for technology selection, UML diagrams, and process modeling. Additionally, it outlines use cases for various user roles, including admins, agriculture field officers, farmers, researchers, and generic users. Furthermore, the chapter includes activity diagrams for key processes and a data model description for the database schema. Together, these elements offer a comprehensive overview of the system's design, emphasizing its potential to revolutionize Sri Lanka's agricultural landscape.

# Chapter 4 – Implementation

## 4.1 Introduction

This chapter includes the implementation process of the project aimed at modernizing data collection in Sri Lanka's agricultural sector. The objective is to create a comprehensive system comprising a mobile application and a cloud-based platform. Transitioning to a cloud-based system improves efficiency by replacing outdated paper methods, enhancing data accuracy, accessibility, and collaboration among stakeholders. The project targets the Sri Lankan agricultural sector focusing on efficient field data collection to facilitate decision-making.

## 4.2 Implementation Methodology

The "Ceylon AgriData" system utilizes a service-oriented multi-tier architecture, which features separate layers for presentation, application logic, and data storage. The presentation layer comprises both a mobile application and a wbapp, developed using Flutter and React, respectively. The application logic is encapsulated in a REST API implemented using Python Flask. Data storage is managed through a MySQL database.

The system integrates these components into a cohesive whole and utilized third-party integrations to enhance the user experience, as detailed in section 4.5. The frontend components are developed independently, while the backend is structured into schemas, models, and routes, with connectivity to the database facilitated by an ORM (Object-Relational Mapping) tool such as SQLAlchemy.

Implementation, resulted in the creation of a mobile application and a React web application designed for agricultural data acquisition and visualization. Detailed user manuals for mobile application can be found in *Appendix* K and UIs of Webapp resides in *Appendix* L.

## 4.3 Implementation Environment

The "Ceylon AgriData" system was implemented on a machine running the Microsoft Windows operating system and its software requirements are summarized in Table 4.1.

### 4.3.1 Front-End Implementation Environment

1. **“Ceylon AgriData” Mobile Application**The mobile application was ingeniously crafted using the Dart programming language in conjunction with the Flutter

framework, enabling the development of natively compiled applications for mobile, web, and desktop from a single codebase.

2. **"Ceylon AgriData" Webapp**

The web component of the project was developed with JavaScript – React framework, leveraging its powerful in-built server capabilities to foster an agile development environment. JavaScript served as the base for building the frontend components, facilitating functionalities such as user input validation, data manipulation, and asynchronous communication with the backend server. The utilization of React's CLI commands, *'$ npm build'* for compiling the application's source code into a production-ready bundle, and *' $ npm start'* for launching a development server.

To facilitate the development and deployment of the frontend components, an environment powered by Node.js and NPM (Node Package Manager) was established. Node.js provided a robust runtime environment for executing JavaScript code outside of a web browser, enabling server-side rendering of React components and efficient dependency management. Meanwhile, NPM served as a crucial tool for installing, managing, and updating the countless of third-party libraries and dependencies required for frontend development. This comprehensive toolset empowered developers to harness the full potential of modern web development practices, ensuring the creation of a sophisticated and responsive user interface tailored to the unique requirements of the agricultural data management system.

The system's diagrams were crafted using Draw.io, an open-source, cross-platform tool renowned for its diagramming capabilities. Meanwhile, Canva, a versatile graphic design platform, was employed to edit and design all images utilized within the mobile application and website, offering a user-friendly interface and a plethora of design options for professional-grade visuals.

### 4.3.2 Back-End Implementation Environment

A virtual machine was set up to host the backend services, with Flask, A lightweight and versatile web application framework, installed to manage the REST API. The command *'$ flask run'* was employed to initiate the Flask development server. Flask served as the backbone of the backend system, providing a lightweight yet powerful foundation for handling HTTP requests and responses. Leveraging Flask's modular design, developers were able to seamlessly

define routes, handle authentication, and integrate various middleware components to enhance functionality and security.

Complementing the Flask framework, Flask-Cors version 4.0.0 played a pivotal role in enabling cross-origin resource sharing (CORS) within the backend system. This crucial extension facilitated seamless communication between the backend server and the frontend client, allowing for the exchange of data across different domains without encountering browser security restrictions. Flask-Cors helps to implement robust API endpoints and ensure smooth interoperability between different kind of components of the agricultural data management system.

To streamline dependency management and package installation within the backend environment, 'Pip' was employed as the primary package manager. Pip facilitated the seamless integration of third-party libraries and extensions, allowing developers to augment the functionality of the Flask framework with ease.

### 4.3.3 Data Persistent and Management Environment

In this project, we used MYSQL as the database for data persistence. MySQL is an open-source database that can handle large volumes of data and can scale to accommodate growing applications with large data amounts, as we expect for AgriData. This is famous for its good community support, reliability and security as a relational database. Further, it is relatively easy to learn, install, configure and manage for developers. In the development environment, WAMP server is used to host the MYSQL database. WAMP server seamlessly connects the database to the backend API, facilitating efficient data exchange. Figure 4.1 illustrates the MYSQL database in WAMP Server.

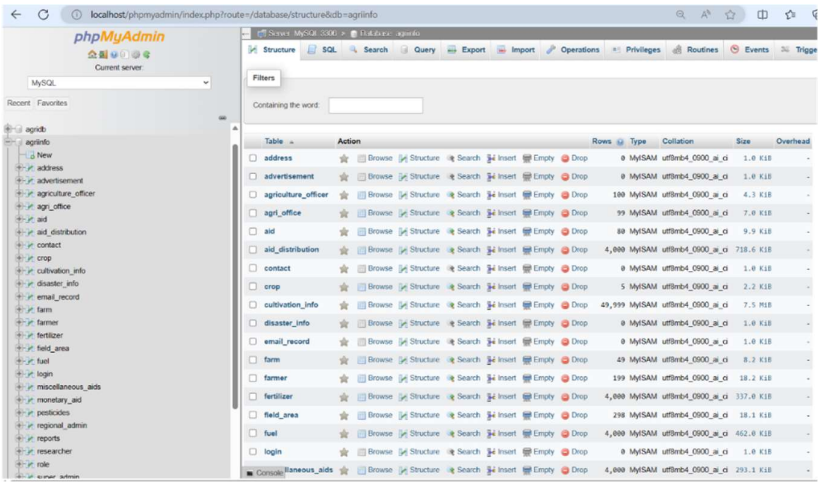



*Figure 4.1: MYSQL database in WAMP Server*

As We use SQLALchemy ORM - Object relational mapping library, a python SQL toolkit that can be used to interact with relational databases using python. It helps to work with databases in an object-oriented way and reduces the complexities of SQL queries to the databases to some extent. One of the main benefits is that supporting multiple databases such as SQLite, MYSQL, Oracle SQL etc. it supports SQL queries using query builder as shown in the code snippet in Figure 4.2 and Figure 4.3. See *Appendix B* for other database models used in the project.

```python
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy import Column, Integer, String, Date, DateTime, Boolean
from sqlalchemy import Column, Integer, String, Date, ForeignKey
from sqlalchemy.orm import relationship

# This file contains the database models for the application.
db = SQLAlchemy()

# The following classes are the database models for the application.
class User(db.Model): # User class
    __tablename__ = 'user' # Table name
    user_id = Column(Integer, primary_key=True) # SQL Alchemy will auto-increment
    first_name = Column(String(100)) # Column name and type
    middle_name = Column(String(100), nullable=True)
    last_name = Column(String(100))
    nic = Column(String(100), unique=True)
    email = Column(String(100), unique=True)
    password = Column(String(100))
    dob = Column(Date)
    role = Column(Integer, nullable=False)
```

*Figure 4.2 :SQL Alchemy used in the project with model class- User*

```python
    # Query the database for the cultivation information
    result = db.session.query(
        AgriOffice.district,
        Crop.crop_name,
        CultivationInfo.cultivation_info_id,
        CultivationInfo.longitude,
        CultivationInfo.latitude,
        AgriOffice.agri_office_id,
    ).join(
        Farm, Farm.farm_id == CultivationInfo.farm_id  # Join with Farm table
    ).join(
        Farmer, Farmer.user_id == Farm.farmer_id  # Join with Farmer table
    ).join(
        AgriOffice, AgriOffice.agri_office_id == Farm.office_id  # Join with AgriOffice table
    ).join(
        Crop, Crop.crop_id == CultivationInfo.crop_id  # Join with Crop table
    ).filter(
        # Apply filters based on the request parameters
        CultivationInfo.agri_year == agri_year,
        CultivationInfo.crop_id == crop_id,
        extract('month', CultivationInfo.estimated_harvesting_date) == month,
        AgriOffice.district == district,
        AgriOffice.agri_office_id == office_id
    ).all()
```

*Figure 4.3: SQL Alchemy query builder example*

It was able to seamlessly integrate it with the Flask web application with SQLite & MYSQL database in this project. SQLite was used only for development purposes in the initial stage of implementation before switching to MYSQL. For Data serialization, we used a python library called "'Marshmallow' a lightweight serialization library, which plays a key role in defining data schemas and transforming complex data structures to and from JSON format in this backend service of the system. See Figure 4.4 for Marshmallow schemas defined in the project backend. See Appendix C for other schemas used in the project.

```python
# This file contains the schemas for the database models
from flask_marshmallow import Marshmallow
# from app import ma
ma = Marshmallow()

# Schemas for the database models
class UserSchema(ma.Schema):
    class Meta:
        fields = ('user_id', 'first_name', 'middle_name', 'last_name', 'email', 'nic', 'dob', 'role')


class FarmSchema(ma.Schema):
    class Meta:
        fields = ('farm_id', 'farm_name', 'address', 'type', 'farmer_id', 'area_of_field', 'owner_nic'

class ContactSchema(ma.Schema):
    class Meta:
        fields = ('contact_id', 'user_id', 'number', 'area_code')
```

*Figure 4.4: Schemas using 'Marshmallow'*

Summarized details regarding the frameworks, programming languages, IDEs, and other essential tools that utilized in the project are provided in Table 4.1. And, the related hardware implementation environment of "Ceylon AgriData" mobile application is as follows in Table 4.2.

*Table 4.1: Summarization of hardware implementation environment of "Ceylon AgriData*

| | **Hardware Environment** |
|---|---|
| **Mobile Application** | • 12th Gen Intel(R) Core (TM) i5-1235U 1.30 GHz, RAM 16.0 GB (15.7 GB usable)<br>• Used, Android Mobile Phone instead of Virtual Emulator. Samsung A30– with Android version 11, 4GB Ram |
| **Website** | • 12th Gen Intel(R) Core (TM) i5-1235U 1.30 GHz, RAM 16.0 GB (15.7 GB usable) |

*Table 4.2: Summarization of software Environment of " Ceylon AgriData" system*

| | Frameworks/ Database Mgt Software | Programming Language | Web Servers | IDE | Other Necessities |
|---|---|---|---|---|---|
| **Mobile Application** | Flutter | Dart | - | Android Studio | Ngrok, Canva Draw.io |
| **Website (Front-End)** | React- version 18.2.0 | JavaScript | React Inbuilt Server (Node.js version 18.16.0 and npm version 9.5.1) | VS Code | Canva Draw.io |
| **REST API (Back-End)** | Flask- version 3.0.0 | Python | Flask Development Server | VS Code | Flask-Cors version 4.0.0 Pip version 19.2.3 |
| **Database** | MySQL version 8.0.31 | Python | Apache version 2.4.54.2 (Wamp Server version 3.3.0 64 bit) | - | SQLAlchemy version 2.0.21 Marshmallow version 3.20.1 |

## 4.4 Utilized Pre-Built Libraries and Frameworks

The project is utilized with pre-built React components and layout structures, as well as Flask libraries and extensions in Python. It also incorporates Material UI and Core UI: UI component libraries for React applications, along with Python Google libraries for various functionalities provided by Google's APIs. Table 4.3 summarizes its description and sources separately.

*Table 4.3: Utilized pre-built libraries and frameworks in "Ceylon AgriData"*

| Category | Description | Source/Reference |
|---|---|---|
| React Template | Pre-built React components and layout structure for web development | Custom development / Open-source libraries |
| Flask Libraries | Libraries and extensions for the Flask web framework in Python | Flask Documentation / PyPI |
| Material UI & CoreUI | UI component libraries for React applications, providing pre-designed components | Material-UI / CoreUI |
| Python Google Libraries | Python libraries provided by Google for various functionalities | Google APIs Python Client Library |
| Flask JWT Token | Library for generating JSON Web Tokens (JWT) for authentication in Flask apps | Flask-JWT-Extended |

## 4.5 Integration of Third-Party Services

The project integrates third-party services to enhance its functionality and user experience. One key integration involves leveraging the Leaflet.js library for interactive mapping on the web, utilizing its API to customize maps and integrate with other web technologies. Additionally, Google APIs, particularly the Gmail API, are utilized for seamless email integration, requiring OAuth 2.0 authentication and API calls integration for secure access and efficient management of email data within the application. See Table 4.4.

*Table 4.4: Third-party services utilized in "Ceylon AgriData" application*

| Third-Party Service | Description | Provider |
|---|---|---|
| Leaflet Mapping | JavaScript library for interactive maps on the web | Leaflet.js (Volodymyr Agafonkin, 2010) |
| Google APIs-Gmail | APIs provided by Google for integrating Gmail functionality into applications | Google (Google Cloud Console) |
| Google Auth Playground | Tool Provided by Google for testing and debugging OAuth authentication flows | Google Developers |

## 4.6 Explanation of Key Code Sections

To elucidate the coding flow in both the mobile application and web application, it is segmented into two sections: the mobile app and the web app. Several major use cases to illustrate this: logging into the system, inserting agriculture data, message broadcasting, representing agricultural data using interactive 'Choropleth Map' of Sri Lanka. Specifically, the requisite backend support with REST API and database integration will be discussed within. For further insights into code explanations, please refer to *Appendix E*.

### 4.6.1 Logging into the system

**1. Mobile application login functionality**

The mobile application is exclusively designed for use by agriculture field officers. The Flutter code snippet for implementing the login functionality is outlined below.

The code snippet of building the "Login Page" UI presents in Figure 4.5(a) and Figure 4.5(b). It constructs a scaffold with a centered body, containing a single child scroll view and padding for layout consistency. Within the scroll view, the UI elements for the login form are organized vertically in a column. These elements include an image asset for the logo, text form fields for username and password entry, an elevated button for submitting the login credentials, and additional UI components for dividing the form sections and displaying options for registration.

```dart
// Build the login page UI
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Center(
      child: SingleChildScrollView(
        child: Padding(
          padding: const EdgeInsets.all(30.0),
          child: Column(
            crossAxisAlignment: CrossAxisAlignment.center,
            children: <Widget>[
              Image.asset(
                "lib/assets/logo.png",
                width: 150.0,
                height: 150.0,
              ),
              const SizedBox(height: 30),
              TextFormField(
                controller: username,
                decoration: const InputDecoration(
                  labelText: "Username",
                  prefixIcon: Icon(Icons.person),
                ),
              ),
              const SizedBox(height: 30),
              TextFormField(
                controller: password,
                obscureText: true,
                decoration: const InputDecoration(
                  labelText: "Password",
                  prefixIcon: Icon(Icons.lock),
                ),
              ),
```

*Figure 4.5 (a):Code snippet of building the "Login Page" UI*

The Login Page UI is in Figure 4.5 (c).


*Figure 4.5 (b): Login Page UI of mobile application*

```
ElevatedButton(
                onPressed:
                    _login, // Call the _login method when the button is pressed
                child: const Text('Login'),
              ),
              const SizedBox(height: 20),
              Row( mainAxisAlignment: MainAxisAlignment.center,
                children: <Widget>[
                  Expanded(
                    child: Divider(
                      color: Colors.teal.shade300,
                      height: 10,
                    ),
                  ),
                  const Text("or"),
                  Expanded(
                    child: Divider(
                      color: Colors.teal.shade300,
                      height: 10,
                    ), ), ], ),
              const SizedBox(height: 20),
              Row( mainAxisAlignment: MainAxisAlignment.center,
                children: <Widget>[
                  const Text(
                    'Don\'t have an account? ',
                    style: TextStyle(fontSize: 16),
                  ),
                  GestureDetector(
                    onTap: () {
                      performRegistration(context);
                    },
                    child: const Text(
                      "Register",
                      style: TextStyle(fontSize: 16, color: Colors.teal),
```

*Figure 4.5 (b): Code snippet of building the "Login Page" UI*

The code snippet in Figure 4.6 (a) and (b) defines a method named "_login()" responsible for handling the login functionality in a Flutter application. Within this method, a constant string "apiUrl" is declared, representing the endpoint URL for the login API. Then, an asynchronous HTTP POST request is sent to this API URL using the "http.post()" method from the "http" package. The request includes headers specifying the content type as JSON, and the request body is encoded into JSON format using "jsonEncode()". The body contains the user's email and password obtained from text form controller "username" and "password". The method executes asynchronously and awaits the response from the API.

Figure 4.7 checks the response status code received from the login API. If the status code is 200 (indicating a successful response), the JSON response body is parsed into a map of string-dynamic pairs using "jsonDecode()". The role value is extracted from this map, along with other user details like token, first name, last name, email, and user ID. Depending on the user's

role, if it equals 4, indicating an agriculture field officer, the application navigates to the home page using “Navigator.pushReplacement()”. If the role is not 4, indicating an invalid login attempt, a toast message displaying "Invalid login" is shown using “Fluttertoast.showToast()”. If the response status code is not 200, indicating a failed login attempt, an alert dialog is displayed with the title "Login Failed" and a message indicating "Invalid username or password", prompting the user to dismiss the dialog by pressing the "OK" button.

```
// Define a method for login functionality
Future<void> _login() async {
  const String apiUrl =
      'https://bluebird-balanced-drum.ngrok-free.app/user/login'; // API Url:Login

  // Send a POST request to the login API
  final response = await http.post(
    Uri.parse(apiUrl),
    headers: <String, String>{
      'Content-Type': 'application/json; charset=UTF-8',
    },
    body: jsonEncode(<String, String>{
      'email': username.text, //username taken by defined text controllers
      'password': password.text, //password taken by defined text controllers
    }),
  );
```

*Figure 4.6: Code snippet of Login function in mobile application*

```
// Check the response status code
if (response.statusCode == 200) {
  // Parse the JSON response
  final Map<String, dynamic> responseData = jsonDecode(response.body);
  // Extract the role value from the JSON response
  final int role = responseData['role'];
  token=responseData['token'];
  firstname=responseData['firstname'];
  lastname=responseData['lastname'];
  email=responseData['email'];
  userid=responseData['user_id'];
  // Check the user role
  if (role == 4) {
    // Redirect to the home page if role is 4
  Navigator.pushReplacement(
    context,
    MaterialPageRoute(builder: (context) => const HomePage()),
  );
```

*Figure 4.7(a): Code snippet of checking response status code and preview relevant notifications*

```
  // Show invalid login message if role is not 4
} else if(role!=4){
  Fluttertoast.showToast(
    msg: 'Invalid login',
    toastLength: Toast.LENGTH_LONG,
    gravity: ToastGravity.BOTTOM,
    timeInSecForIosWeb: 1,
    backgroundColor: Colors.transparent,
    textColor: Colors.red,
    fontSize: 16.0,
  );
}
}else{
// Show error message for failed login
showDialog(
  context: context,
  builder: (BuildContext context) {
    return AlertDialog(
      title: const Text('Login Failed'),
      content: const Text('Invalid username or password.'),
      actions: <Widget>[
        TextButton(
          onPressed: () {
            Navigator.of(context).pop();
          },
          child: const Text('OK'),
        ),
```

*Figure 4.7(b): Code snippet of checking response status code and preview relevant notifications*

When the mobile application sends a POST request to the login API, it triggers the corresponding endpoint in the REST API. The relevant code snippet for this API endpoint is provided in Figure 4.8(a) and (b).

```
//defines a route
app.register_blueprint(user_routes, url_prefix='/user')
///
@user_routes.route("/login", methods=['POST'])
def login():
    if request.is_json:
        email = request.json['email']
        password = request.json['password']
    else:
        email = request.form['email']
        password = request.form['password']

    user = User( email=email, password=password)
    user = user_login(user)
    userEmail=user.email
    userFname=user.first_name
    userLname=user.last_name
    user_id=user.user_id
```

*Figure 4.8 (a): Code snippet for Login endpoint of REST API*

```
    if user:
        access_token = get_access_token(user)
        #access_token = create_access_token(identity=user.user_id, expires_delta=timedelta(days=1))
        return jsonify(message="Login succeeded!", token=access_token, role=user.role, firstname=userFname,
                       lastname=userLname,email=userEmail, user_id=user_id), 200
    else:
        return jsonify(message='error in email or password', result=user), 401

///
def  user_login(user):
    user=User.query.filter_by(email=user.email, password=user.password).first()
    return user
///
def get_access_token(user):
        access_token = create_access_token(identity=user.user_id, expires_delta=timedelta(days=1))
        return access_token
```

*Figure 4.8 (b): Code snippet for Login endpoint of REST API*

When a POST request is made to this route "/user/login", the function "login()" is executed. The function checks if the request is in JSON format or form data and extracts the email and password accordingly. It then attempts to authenticate the user by querying the database with the provided credentials using the "user_login()" function. If authentication is successful, an access token is generated using "create_access_token()" function, and a JSON response is returned with a success message, access token, user role, first name, last name, and user ID. If authentication fails, a JSON response with an error message and HTTP status code 401 (Unauthorized) is returned. The "user_login()" function queries the database for a user with the provided email and password and returns the user object if found. Finally, the "get_access_token()" function generates an access token for the authenticated user, which is then returned.

1. **Web application Login Functionality**

Code snippets pertaining to the frontend of the web application are detailed below. See Figure 4.9(a) for the UI in Web app. The React code snippet for implementing the login functionality is provided in Figure 4.9(b). This code snippet represents a modal component for user login functionality in a web application built with React. The modal includes input fields for the user's email address and password, along with buttons for signing in and registering. The "show" and "handleClose" props control the visibility of the modal and its closing action. The user input for email and password is captured using the "Form.Control" component and stored in the "username" and "password" state variables, respectively. The "password" function is called when the "Sign In" button is clicked, while the "handleRegister" function is called when

the “Register” button is clicked. Figure 4.10 shows the modal built from the above explained code snippet.

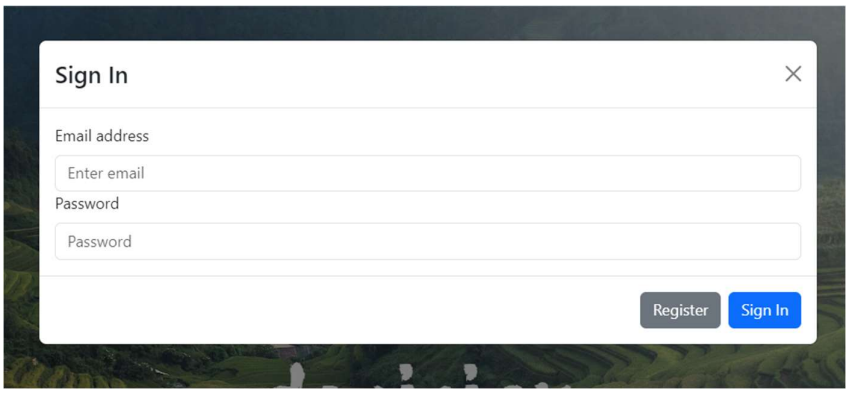



*Figure 4.9(a): UI modal of Login in web application*

```
//Render the LoginModal component with show and handleClose props
<LoginModal show={show} handleClose={() => setShow(false)} />

//Modal header with a close button and title
<Modal.Header closeButton>
          <Modal.Title>Sign In</Modal.Title>
        </Modal.Header>
//Modal body containing a form for email and password input
        <Modal.Body>
          <Form>
            <Form.Group controlId="formBasicEmail">
              <Form.Label>Email address</Form.Label>
              <Form.Control
                type="email"
                placeholder="Enter email"
                value={username}
                onChange={(e) => setUsername(e.target.value)}
              />
            </Form.Group>

            <Form.Group controlId="formBasicPassword">
              <Form.Label>Password</Form.Label>
              <Form.Control
                type="password"
                placeholder="Password"
                value={password}
                onChange={(e) => setPassword(e.target.value)}
              />
            </Form.Group>
          </Form>
        </Modal.Body>
//Modal footer containing buttons for registration and signing in
        <Modal.Footer>
          <Button variant="secondary" className="ml-auto" onClick={handleRegister}>
            Register
          </Button>
          <Button variant="primary" onClick={handleSubmit}>
            Sign In
          </Button>
        </Modal.Footer>
      </Modal>
```

*Figure 4.9(b): Code snippet of building login modal in web application (front end)*

Figure 4.10 shows the code snippet of the login function in web application front end. It verifies if the username and password fields are filled, then sends a POST request to a login endpoint with the provided credentials. If successful, it validates the user with the received token for further authentication.

```
// Function to handle user login
  const logInUser = () => {
    // Check if the username field is empty
    if (username.length === 0) {
      alert('Email has left Blank!')
    // Check if the password field is empty
    } else if (password.length === 0) {
      alert('password has left Blank!')
    // If both fields are filled, send a POST request to the login endpoint
    } else {
      axios
        .post('http://127.0.0.1:5000/user/login', {
          email: username,
          password: password,
        })
        .then(function (response) {
          console.log(response)
          // If a token is received in the response, validate the user with the token
          if (response.data.token) {
            // Validate the user with the token
            axios
              .post(
                'http://127.0.0.1:5000/user/validate',
                {
                  email: username,
                },
                {
                // Include the token in the request headers for authentication
                  headers: {
                    Authorization: `Bearer ${response.data.token}`,
                  },
                },
              )
```

*Figure 4.10: Code snippet of login functionality in web application front end*

When the API sends a response, the frontend of the web application validates the response status to display the relevant messages. This functionality is implemented in the code snippet shown in Figure 4.11(a) and Figure 4.11(b). User validation code snippet is in Figure 4.12.

```
.then(function (validateResponse) {
                    // Log the validation response
                console.log(validateResponse)
                    // Check if the user is valid
                if (validateResponse.data.valid) {
                        // Store the token in local storage
                  localStorage.setItem('token', response.data.token)
                        // Set isValidUser state to true
                  setIsValidUser(true)
                        // Set the response state
                  setResponse(validateResponse)
                } else {
                        // Show an alert if user validation failed
                  alert('User validation failed, please log in again or contact Admin')
                }
              })
```

*Figure 4.11 (a) :Code snippet of validating the API response*

```
useEffect(() => {
    // Check if isValidUser and response exist and have data
  if (isValidUser && response && response.data) {
        // Check the user role and navigate accordingly
    if ([1].includes(response.data.user.role)) {
            // Navigate to dashboard for certain user roles
      navigate('/dashboard', { state: { isValidUser: isValidUser } }) // Update isValidUser to true
    } else if ([2, 5, 6].includes(response.data.user.role)) {
            // Navigate to genericDashboard for certain user roles
      navigate('/genericDashboard', { state: { isValidUser: isValidUser } }) // Update isValidUser to true
    } else if ([3, 4].includes(response.data.user.role)) {
            // Navigate to officer-dashboard for certain user roles
      console.log(isValidUser, 'isValidUser')
      navigate('/officer-dashboard', { state: { isValidUser: isValidUser } }) // Update isValidUser to true
    } else {
            // Show an alert for failed logging
      alert('Logging was failed, please log in again or contact Admin')
    }
  }
}, [isValidUser, response])
```

*Figure 4.11(b):User validation code snippet of webapp (front end)*

```
      .catch(function (validateError) {
        console.log(validateError, 'validateError')
            // Show an alert for error validating token
        alert('Error validating token')
      })
  } else {
      // Show an alert for invalid token
    alert('Invalid token')
  }
})
.catch(function (error) {
  console.log(error, 'error')
    // Check if there is a response error
  if (error.response) {
        // Check if the status code is 401 (Unauthorized)
    if (error.response.status === 401) {
            // Show an alert for invalid credentials
      alert('Invalid credentials, please try again.')
    }
      // Check if the error is an AxiosError
  } else if (error instanceof AxiosError) {
        // Show an alert for network error
    alert('An network occurred. Please try again later')
      // Check if the error code is ECONNREFUSED (Connection refused)
  } else if (error.code === 'ECONNREFUSED') {
        // Show an alert for connection refused
    alert('Connection refused. Please check your network connection.')
  } else {
    console.log(error)
        // Show an alert for general error
    alert('An error occurred. Please try again later.')
  }
})
```

*Figure 4.12: Code snippet of validating the API response*

### 4.6.2 Inserting agriculture data into the system

The primary categories of agricultural data to be incorporated into the system include details regarding farmers, farms, cultivations, aid distributions, and disasters. Among these, the code snippets pertaining to "adding cultivation information" are integrated within following sections.

Refer *Appendix D, E and F* for selected coding snippets of mobile application, web application and REST API of the project.

**1. Add cultivation information through mobile application**

Code snippet represents a user interface in Flutter for adding cultivation details is in Figure 4.14. It consists of various input fields for entering information such as farm ID, location details, crop details, cultivation area, estimated harvest, dates, agri year, quartile, and net yield. Users can also retrieve location details and select dates using date pickers. Upon filling out the required fields, users can submit the form, which triggers the function "_performLogCultivation(). Additionally, users can navigate back to the home page using the home button on the app bar. Thus built UI is in Figure 4.13.

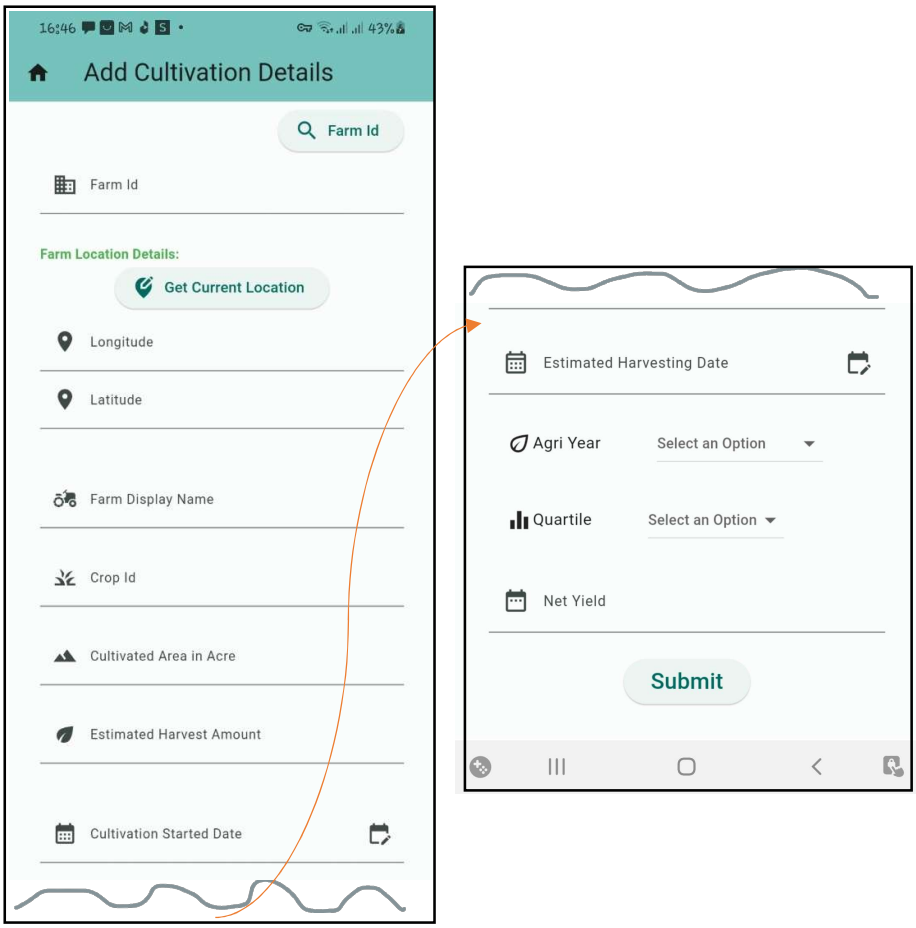



*Figure 4.13:UI for add cultivation details in mobile application*

```
//AppBar
appBar: AppBar(
  title: const Text(
    "Add Cultivation Details", // Title of the app bar
    style: TextStyle(fontSize: 30.0), // Styling for the title text
  ),
  backgroundColor: Colors.teal.shade200,// Background color of the app bar
  leading: IconButton(
    icon: const Icon(Icons.home),// Icon for navigating back to home page
    onPressed: () {
      Navigator.pushReplacement(// Replacement navigation to home page
        context,
        MaterialPageRoute(builder: (context) => const HomePage()),// Builds HomePage widget
      );
///codes relating to body …
body: SingleChildScrollView(// Widget for scrollable content
  child: Padding(// Widget for padding around the content
    padding: const EdgeInsets.all(30.0),// Padding for all sides
    child:
        Column(crossAxisAlignment: CrossAxisAlignment.center, children: [
      Row(
        mainAxisAlignment: MainAxisAlignment.end,
        children: [
          ElevatedButton.icon(// Elevated button widget with icon and label
            icon: const Icon(Icons.search), // Add the search icon here
            label: const Text(
              'Farm Id',
              style: TextStyle(fontSize: 18.0),
            ),
            onPressed: () {
              _performFarmIdSearch(context);// Functionality for performing farm ID search
            },
          ),
        ],
///repeats this…
//calling function for API all by pressing submit button
ElevatedButton(// Elevated button widget for submitting cultivation details
  onPressed: () {// Functionality to be executed when the button is pressed
    _performLogCultivation();// Calls the function to log cultivation details
  },
  child: const Text(
    'Submit',
    style: TextStyle(fontSize: 25.0),
  ),
```

*Figure 4.14:Code snippet represents a user interface in Flutter for adding cultivation details*

**Get Current location**

This code in Figure 4.15 creates a button labeled "Get Current Location" with an edit location icon. When pressed, it calls the _performGetLocation(context) function to retrieve the current location. The code in Figure 4.16 defines that asynchronously retrieves the current location. It opens a new screen using to get the location details. After receiving the result from the new screen, it updates the latitude and longitude text fields with the obtained.

"GetCurrentLocation" in Figure 4.16 utilizes the geolocator and geocoding packages to fetch the location coordinates and address details. See *Appendix G* for more information. The screen includes a Google Map widget to visualize the location, buttons to fetch the current location and confirm it, and text widgets to display latitude, longitude, and address information in Figure 4.17.

```
ElevatedButton.icon(
icon: const Icon(
    Icons.edit_location_alt), // Add the search icon here
label: const Text(
  'Get Current Location',
  style: TextStyle(fontSize: 18.0),
),
onPressed: () {
  _performGetLocation(context);
},
),
```

*Figure 4.15: Code snippet for get current location button in frontend UI in mobile application*

```
void _performGetLocation(BuildContext context) async {
  final result = await Navigator.push(
    context,
    MaterialPageRoute(
      builder: (context) => const GetCurrentLocation(),
    ),
  );
  print(result.toString());
  if (result != null) {
    setState(() {
      latitude.text = result['latitude'].toString();
      longitude.text = result['longitude'].toString();
    });
  }
}
```

*Figure 4.16: Code sippet regarding the function that gets location*

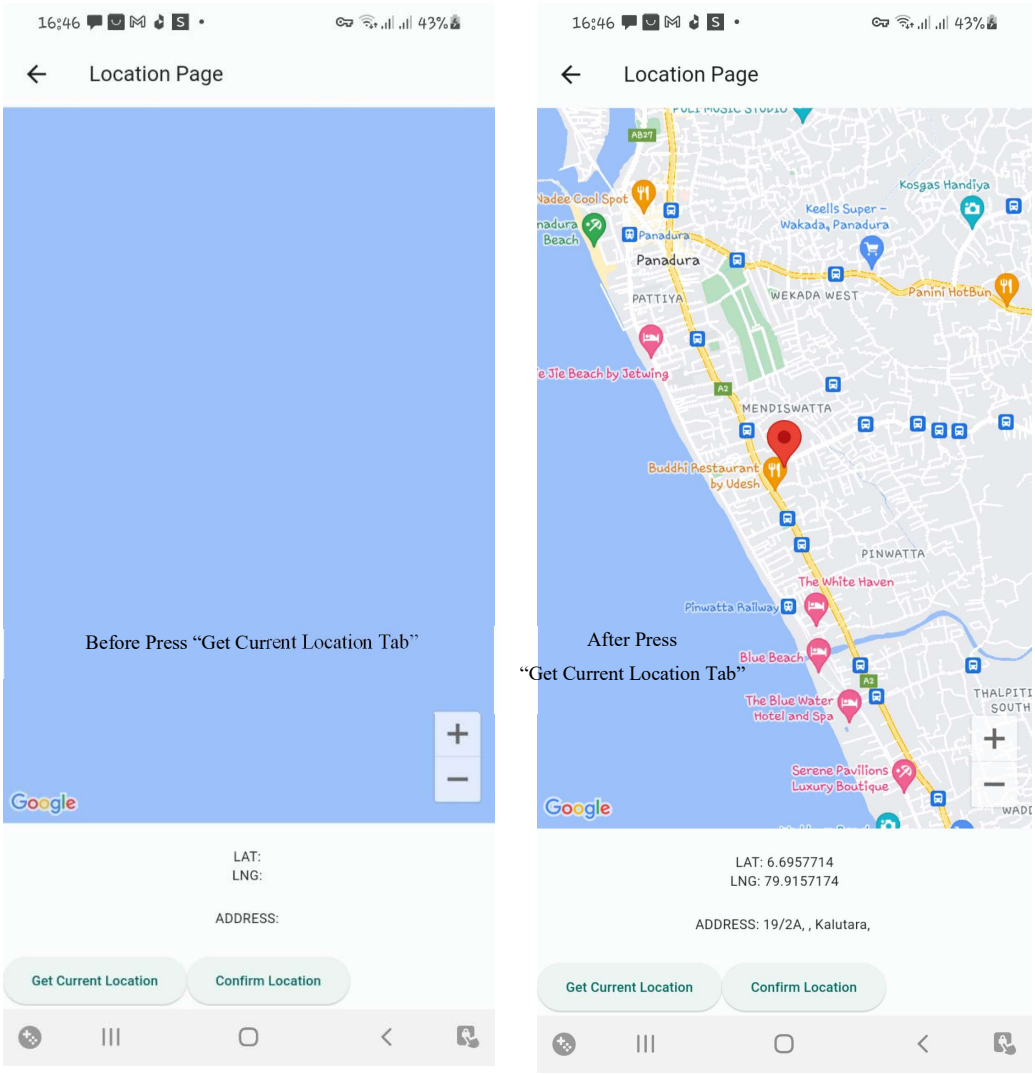



*Figure 4.17: UIs of get current location functionality in mobile application*

When press the "Submit" button in Figure 4.13, it asynchronously sending a POST request to a specific API endpoint to log cultivation details. It begins by defining the URL of the API. Using the "http" package, it constructs and sends the POST request to the designated endpoint, incorporating necessary headers like content type and authorization token. The body of the request is formed by encoding cultivation details such as farm ID, crop ID, display name, cultivation start date, estimated harvesting date, longitude, latitude, area of cultivation, estimated harvest, agricultural year, and quarter as JSON. Before sending, it ensures appropriate data types by parsing relevant text field inputs. After the request is made, the function awaits the response. This process facilitates the inserting of cultivation information via an external API. The relevant code snippet is in Figure 4.18.

```
Future<void> _performLogCultivation() async {
  // API URL for cultivation information
  const String apiUrl =
      'https://bluebird-balanced-drum.ngrok-free.app/cultivation/info';

  // Sending POST request to the API with cultivation information
  final response = await http.post(
    Uri.parse(apiUrl),
    headers: <String, String>{
      'Content-Type': 'application/json; charset=UTF-8',
      'Authorization': 'Bearer $token', // Authorization token
    },
    body: jsonEncode(<String, dynamic>{
      'farm_id': int.parse(farmId.text),
      'crop_id': int.parse(cropId.text),
      'display_name': displayName.text,
      'started_date': cultivationStartedDate.text,
      'estimated_harvesting_date': estimatedHarvestingDate.text,
      'longitude': longitude.text,
      'latitude': latitude.text,
      'area_of_cultivation': int.parse(cultivationAreaInAcre.text),
      'estimated_harvest': int.parse(estimatedHarvest.text),
      "agri_year": agri_year,
      "quarter": quarter
    }),
  );
```

*Figure 4.18: Code snippet of sending post request to API to insert cultivation information from mobile application front-end*

The **authorization token** is used for authentication, ensuring that only authorized users can access the cultivation information endpoint. It validates the identity of the client making the request.

Figure 4.19 showcase the code snippet of Flask route for adding cultivation information. It requires JWT authentication. It extracts data from the request, checks if farm and crop IDs exist, creates a new cultivation record, adds it to the database, and returns a success message.

```python
# Route for adding cultivation information
@cultivation_routes.route('/info', methods=['POST'])
@jwt_required()# Requires JWT authentication
def add_CultivationInfo():
    # Get the request data
    data = request.get_json()

    # Check if farm_id exists in the Farm table
    farm = Farm.query.get(data['farm_id'])
    if not farm:
        return jsonify(message='Invalid farm_id'), 400

    # Check if crop_id exists in the Crop table
    crop = Crop.query.get(data['crop_id'])
    if not crop:
        return jsonify(message='Invalid crop_id'), 400

    # Create a new cultivation info record
    cultivation_info = CultivationInfo(
        display_name=data['display_name'],
        farm_id=data['farm_id'],
        crop_id=data['crop_id'],
        longitude=data['longitude'],
        latitude=data['latitude'],
        area_of_cultivation=data['area_of_cultivation'],
        started_date=parse_date(data['started_date']),
        estimated_harvesting_date=parse_date(data['estimated_harvesting_date']),
        estimated_harvest=data['estimated_harvest'],
        agri_year=data['agri_year'],
        quarter=data['quarter'],
        added_by=get_jwt_identity(),
        updated_by=get_jwt_identity(),
        added_date=datetime.date.today()
    )

    # Add the cultivation info record to the database
    db.session.add(cultivation_info)
    db.session.commit()

    # Return a response
    return jsonify(message='Cultivation info added successfully'), 201
```

*Figure 4.19: Code snippet of flask route of add cultivation information in REST API*

**2. Manage cultivation information through web application**

The web application utilizes functions for inserting, updating, searching, and deleting cultivation information. The Figures below display the relevant code snippets and UI elements in the web application. See *Appendix E* for other selected agricultural data management code snippets.

Figure 4.20 showcases the code snippet relevant to Tab pane, Figure 4.21. The code presents a tabbed interface with three panes: one for adding new cultivation information, another for searching existing information, and a third for updating details. This setup streamlines user interaction, offering distinct functionalities for managing cultivation data efficiently within a web application.

```
<CTabContent>
        {/* First tab pane for adding cultivation info */}
       <CTabPane role="tabpanel" aria-labelledby="home-tab-pane" visible={activeKey === 1}>
        <AddCultivationInfo />
       </CTabPane>
        {/* Second tab pane for searching cultivation info */}
       <CTabPane role="tabpanel" aria-labelledby="profile-tab-pane" visible={activeKey === 2}>
        <SearchCultivationInfo />
       </CTabPane>
        {/* Third tab pane for updating information */}
       <CTabPane role="tabpanel" aria-labelledby="disabled-tab-pane" visible={activeKey === 4}>
        <UpdateInformation />
       </CTabPane>
     </CTabContent>
```

*Figure 4.20: Code snippet of tab pane in manage cultivation information in web application*

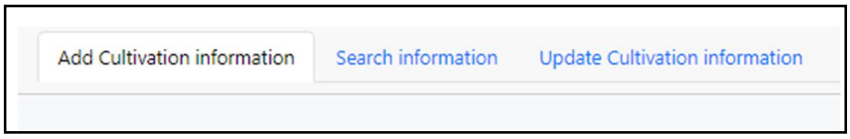



*Figure 4.21: Tab pane of manage cultivation information functionality in web application*

Each functionality: Add, search, and update-utilizes a designed form. The code snippet for a single input text is depicted in Figure 4.22, while similar code structures for other functionalities are present throughout. Specifically, Figure 4.23 illustrates the forms used for managing the "Add Cultivation" functionalities.

```
{/*
  This JSX code represents an input group component with conditional styling based on whether the form is empty
  The input group contains a text label "Farm ID" and an input field for entering the farm ID.
  The input field is disabled to prevent user modification.
*/}
                <CInputGroup className={`mb-3 ${isFormEmpty ? 'border border-danger' : ''}`}>
                  <CInputGroupText>Farm ID</CInputGroupText>
                  <CFormInput
                    placeholder="Farm ID"
                    autoComplete="Farm ID"
                    onChange={handleInputChange}
                    name="farm_id"
                    value={farm.farm_id}
                    disabled
                  />
                </CInputGroup>
```

*Figure 4.22: The code snippet for a single input text in web application*

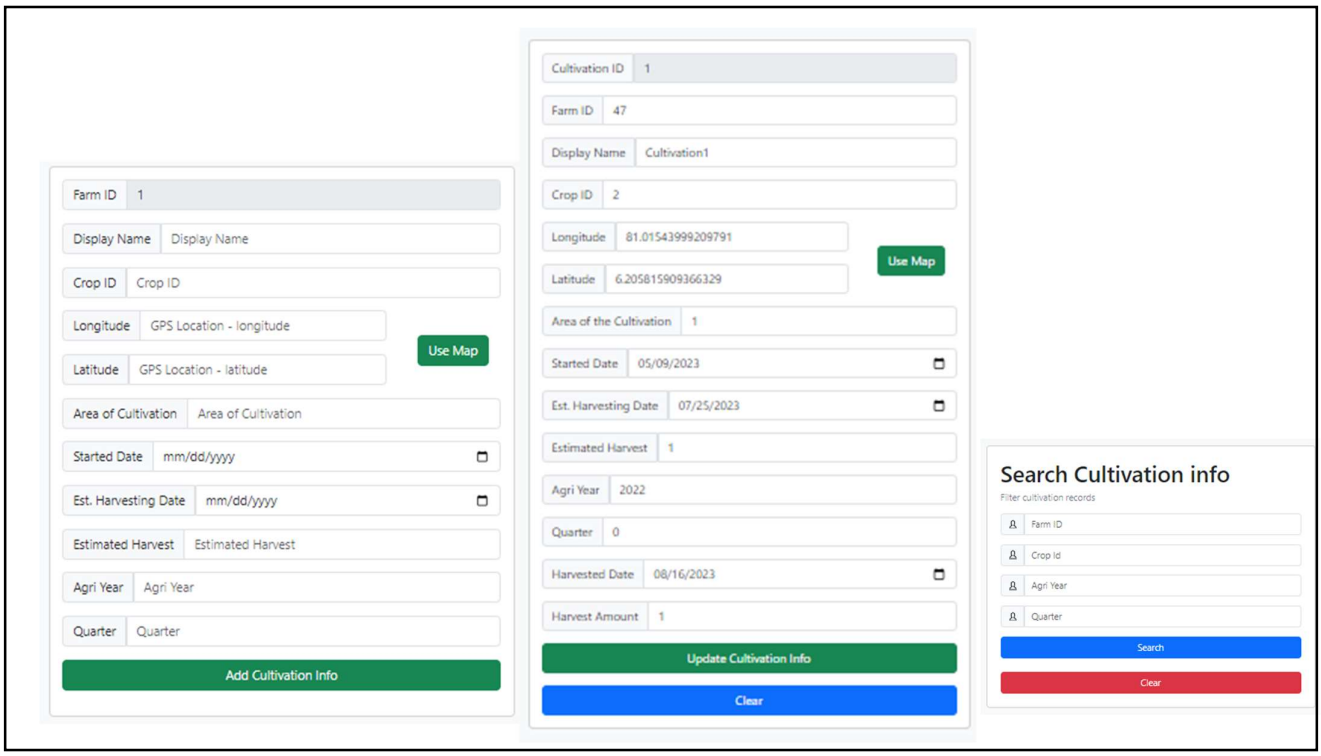



*Figure 4.23:forms used for managing the "Add Cultivation" functionalities in web application*

After filling the form, when *'Add Cultivation Info'* button pressed the *'handleSubmit'* function is triggered. It prevents the default form submission behavior, checks if any form field is empty or null, and displays an alert message if any field is empty. If all fields are filled, it attempts to add cultivation information by calling the *'addCultivationInfo'* function asynchronously. Depending on the response status, it sets a state variable to indicate successful addition or displays an error message. The related code snippet is in Figure 4.24.

```
const handleSubmit = async (event) => {
    console.log(formData)
    if (event) {
      event.preventDefault()

      if (Object.values(formData).some((value) => value === '' || value === null)) {
        setIsFormEmpty(true)
        alert('Please fill in all the fields.')
        return
      } else {
        try {
          const response = await addCultivationInfo(formData)
          console.log(response)
          if (response.request.status === 201) {
            setCultInformation(true)
            alert('Cultivation information added successfully!')
          } else {
            alert(response, 'Error occurred while adding cultivation information.')
          }
        } catch (error) {
          console.error(error)
          alert('Error occurred while adding cultivation information.')
        }
      }
    }
  }
```

*Figure 4.24: Handle submit function for calling 'Add Cultivation Information Function' that sends request to REST API*

Figure 4.25 shows code snippet of making a POST request to the specified endpoint with the provided form data. It first retrieves the authentication token from local storage, then sends the POST request with the form data and includes the token in the request headers for authentication purposes. If the request is successful, it returns the response object. If an error occurs during the request, it checks the status code of the error response. If the status code is 400, it displays an alert message with the error message from the response data. Otherwise, it logs the error to the console.

```
import axios from 'axios'

const addCultivationInfo = async (formData) => {
  const token = localStorage.getItem('token')
  try {
    const response = await axios.post(`http://127.0.0.1:5000/cultivation/info`, formData, {
      headers: {
        Authorization: `Bearer ${token}`,
      },
    })
    return response
  } catch (error) {
    if (error.response.status === 400) {
      alert(error.response.data.message)
      return
    }
    console.error('Failed to fetch farm details:', error)
    return
  }
}
```

*Figure 4.25: Code snippet of making a POST request to the specified endpoint*

Update, delete, and search endpoints are invoked from the web application frontend using a similar approach as described earlier. Below are the code snippets for these endpoints in the REST API in Table 4.5.

*Table 4.5: code snippets for Update, delete, and search endpoints in the REST API*

| Code Snippet | Description |
|---|---|
| **Search** | |

```python
@cultivation_routes.route('/search', methods=['GET'])
@jwt_required()
def search_CultivationInfo():
    # Get the filter parameters
    farm_id = request.args.get('farm_id')
    crop_id = request.args.get('crop_id')
   //--rest parameters--//
    # Build the filter conditions
    filter_conditions = []
    if farm_id:
        filter_conditions.append(CultivationInfo.farm_id == farm_id)
    //--other filters--//
    # Search for cultivation info records based on the filters
    query = CultivationInfo.query.filter(
        *filter_conditions)
   #Pagination
    pagination = query.paginate(page=page, per_page=per_page)
    cultivation_info = pagination.items
    # Return the search results
    result = cultivation_infos_schema.dump(cultivation_info)
    return jsonify({
        'data': result,
        'total_pages': pagination.pages,
        'current_page': pagination.page,
        'per_page': pagination.per_page,
        'total_items': pagination.total
    }), 200
```

**Description**

- Defines an endpoint for updating cultivation information
- Receives a PUT request with the ID of the cultivation information to be updated.
- Checks if the record exists and if the provided farm and crop IDs are valid.
- If everything is valid, it updates the cultivation information record with the provided data and commits the changes to the database.
- Also handles optional fields like harvested date and amount if provided

**Update**

```
@cultivation_routes.route('/<int:cultivation_id>', methods=['PUT'])
@jwt_required()
def update_CultivationInfo(cultivation_id):
 # Get the request data
    data = request.get_json()
# Check if the cultivation info record exists
    cultivation_info = CultivationInfo.query.get(cultivation_id)
    if not cultivation_info:
        return jsonify(message='Cultivation info not found'), 404
# Check if farm_id exists in the Farm table
farm = Farm.query.get(data['farm_id'])
    if not farm:
        return jsonify(message='Invalid farm_id'), 400
crop = Crop.query.get(data['crop_id'])
    if not crop:
        return jsonify(message='Invalid crop_id'), 400
# Update the cultivation info record
    cultivation_info.display_name = data['display_name']
        --//rest updates//--
# Commit the changes to the database
    db.session.commit()
#Return a response
    return jsonify(message='Cultivation info updated successfully'), 200
```

**Description**

- Defines an endpoint for searching cultivation information based on various filter parameters such as farm ID, crop ID, agricultural year, and quarter.
- Receives a GET request with optional query parameters and constructs filter conditions based on the provided parameters.
- Searches for cultivation information records matching the filter conditions.
- Paginates the results, and returns them along with pagination metadata.

**Delete**

```
@cultivation_routes.route('/<int:cultivation_id>', methods=['PUT'])
@jwt_required()
def update_CultivationInfo(cultivation_id):
 # Get the request data
    data = request.get_json()
# Check if the cultivation info record exists
    cultivation_info = CultivationInfo.query.get(cultivation_id)
    if not cultivation_info:
        return jsonify(message='Cultivation info not found'), 404
# Check if farm_id exists in the Farm table
farm = Farm.query.get(data['farm_id'])
    if not farm:
        return jsonify(message='Invalid farm_id'), 400
crop = Crop.query.get(data['crop_id'])
    if not crop:
        return jsonify(message='Invalid crop_id'), 400
# Update the cultivation info record
    cultivation_info.display_name = data['display_name']
        --//rest updates//--
# Commit the changes to the database
    db.session.commit()
#Return a response
    return jsonify(message='Cultivation info updated successfully'), 200
```

**Description**

- Defines an endpoint for deleting cultivation information.
- Receives a DELETE request with the ID of the cultivation information to be deleted.
- Checks if the record exists, and if so, it deletes the record from the database and commits the changes.
- Returns a success message upon successful deletion or a message indicating that the record was not found if it does not exist.

**Get cultivation location  using Map in web application**

When the "Use Map" button (in Figure 4.23) is clicked, the click event is handled by the function "handleSetGPSLocation" with the parameter set to true. Inside this event handler function, the state variable "isMapOpen" is changed using the "setOpenMap" useState function in React. Subsequently, as "isMapOpen" is set to true, the map view of the "SelectArea" component is opened. Within the "SelectArea" component, the map is loaded using React Leaflet components, allowing the user to select a location using a marker icon as in Figure 4.29. After the user clicks on the save button, the coordinates are saved in the local storage

(Evidence in Figure 4.27). Finally, when the user clicks on the close button, the map view is closed, and the saved coordinates are set to the "formData" state coordinates fields in the "handleSetGPSLocation" function. Consider the code snippet in Figure 4.28 (a) and Figure 4.28(b).

```
//UseMap Button
<CButton color="success" onClick={() => handleSetGPSLocation(true)}>
    Use Map
</CButton>

//Event handler function
const handleSetGPSLocation = (value) => {
    setOpenMap(value)
    if (!value) {
//Retrieve coordinates from localStorage and update formData state
      const coordinates = JSON.parse(localStorage.getItem('coordinates'))
      setFormData((prevFormData) => ({
        ...prevFormData,
        latitude: coordinates[0],
        longitude: coordinates[1],}))
      setSelectedCoordinates(true)}}

//close map view
<SelectArea userCoordinates={selectedCoordinates} />
<div className="col-sm-2 mb-2 d-flex align-items-center justify-content-center">
   <CButton color="danger" onClick={() => handleSetGPSLocation(false)}> Close</CButton>
      </div>

//selectArea component
const SelectArea = () => {
  const [coordinates, setCoordinates] = useState([7.505, 80.35])
  const customIcon = new Icon({
    iconUrl: 'https://cdn-icons-png.flaticon.com/128/8326/8326599.png',
    iconSize: [38, 38],
  })
```

*Figure 4.28(a): Code snippets of getting longitude and lattitude of cultivatin location in web application using leaflet*

```
//Evet handler for map click
  const MapClickHandler = () => {
    const map = useMap()
    useMapEvents({
      click: (e) => {
//update coordinates on map click
        setCoordinates([e.latlng.lat, e.latlng.lng])
        console.log('resize', coordinates)
        map.invalidateSize()},})
    return null}

//Function to save coordinates to localStorage
  const handleConfirm = () => {
    localStorage.setItem('coordinates', JSON.stringify(coordinates))}
  return (
    <div>
      <h1>Select Map Area</h1>
//react leaflet map container
      <MapContainer center={coordinates} zoom={30} style={{ height: '400px', width: '100%' }}>
        <TileLayer url="https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png" />
//Handler for map click
<MapClickHandler />
//marker to select coordinates
        <Marker position={coordinates} icon={customIcon} />
        </MapContainer>
//button to save coordinates
        <CButton className="m-2" onClick={handleConfirm}>
        Save Coordinates
       </CButton>
    </div>
```

*Figure 4..28(b): Code snippets of getting longitude and lattitude of cultivatin location in web application using leaflet*

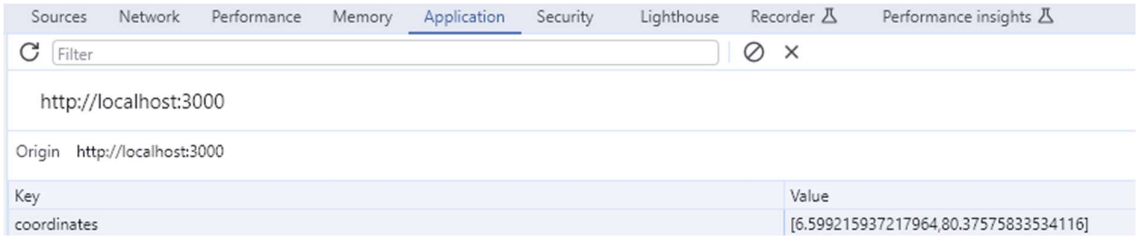
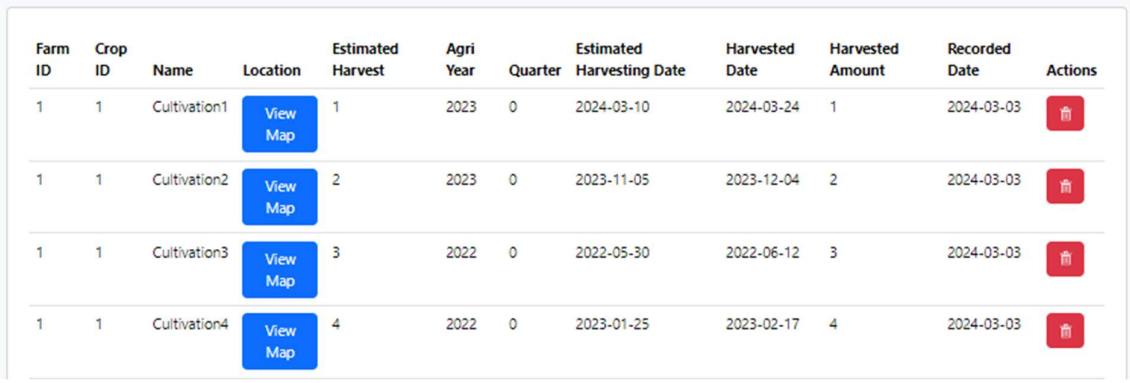



*Figure 4.27: Local storage saves the selected coordinates*

**Viewing the map**

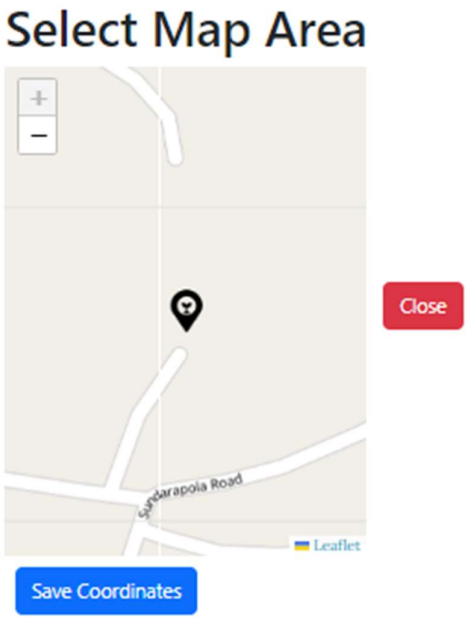
In the **search result** data cell, when the user clicks on the "View Map" button as depicted in Figure 4.29, the event handler function alters the state of "isMapOpen" and sets the relevant coordinates in the local storage for use in the ShowMap component. Subsequently, the ShowMap component is rendered within the search form using an if condition and, previewed

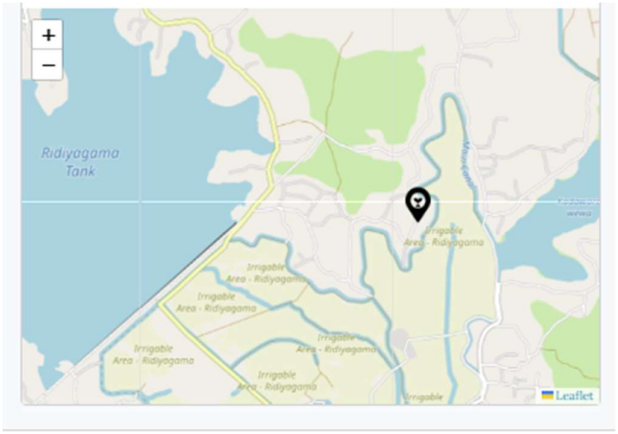
map container is illustrated in Figure 4.30. The related code snippets are presented in Figure 4.31.



| Farm ID | Crop ID | Name | Location | Estimated Harvest | Agri Year | Quarter | Estimated Harvesting Date | Harvested Date | Harvested Amount | Recorded Date | Actions |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | Cultivation1 | View Map | 1 | 2023 | 0 | 2024-03-10 | 2024-03-24 | 1 | 2024-03-03 | |
| 1 | 1 | Cultivation2 | View Map | 2 | 2023 | 0 | 2023-11-05 | 2023-12-04 | 2 | 2024-03-03 | |
| 1 | 1 | Cultivation3 | View Map | 3 | 2022 | 0 | 2022-05-30 | 2022-06-12 | 3 | 2024-03-03 | |
| 1 | 1 | Cultivation4 | View Map | 4 | 2022 | 0 | 2023-01-25 | 2023-02-17 | 4 | 2024-03-03 | |

*Figure 4 .29: Preview of search results in search cultivation information in web application*




*Previewed map container when "View Map" , clicked in web application*

*Figure 4.26: Preview map when "Use Map" button is clicked in add cultivation information functionality in web application*

```
//handles using handler function…
{isMapOpen ? (
    <ShowMap />
        ) : (//other part of code..){if condition}

------//ShowMap Starts//------
const ShowMap = () => {
// Retrieve default longitude and latitude from localStorage
  const defaultLongitude = parseFloat(localStorage.getItem('longitude'))
  const defaultLatitude = parseFloat(localStorage.getItem('latitude'))

// set coordinates based on retrieved longitude and lattitude
 const coordinates = [defaultLatitude, defaultLongitude]

//Log longitude and latitude from local storage
 console.log('Longitude from local storage:', defaultLongitude)
 console.log('Latitude from local storage:', defaultLatitude)

//define custom marker icon
  const customIcon = new Icon({
    iconUrl: 'https://cdn-icons-png.flaticon.com/128/8326/8326599.png',
    iconSize: [38, 38],
  })

  return (
    <div>
      <h1>Select Map Area</h1>
//react leaflet map container
      <MapContainer center={coordinates} zoom={30} style={{ height: '400px', width: '100%' }}>
        <TileLayer url="https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png" />
        <Marker position={coordinates} icon={customIcon} />
      </MapContainer>
    </div>
  )
}

export default ShowMap
```

*Figure 4.29: Preview map when "Use Map" button is clicked in add cultivation information functionality in web application*

### 4.6.3 Generating Reports

Different types of reports are available for generation within the "Ceylon AgriData" system. Among these options, report generating of cultivation information is discussed here. For a comprehensive list of reports that can be generated, refer to *Appendix A*.

**Generating reports on cultivation information functionality**

1. **Report 1: Estimated harvest vs Actual harvest**

The code snippet in Figure 4.30, renders a card body that facilitates the display and interaction with harvest data pertaining to a specified time range. It begins with a title indicating the total harvest within the defined period, followed by a textual representation of the start and end dates of the selected timeframe obtained from the 'formData' state. A dropdown menu allows users to select the year, triggering the 'handleYearChange' function upon selection change. The centerpiece of the component is a bar chart ('CChartBar') presenting expected and actual harvest amounts for various crops. These datasets are dynamically populated using the 'harvestData' state, with each bar representing a different crop. The chart's colors distinguish between expected and actual harvest amounts, offering users a clear visual representation of cultivation performance over time. Figure 4.31 showcases the output of the above-described code snippet.

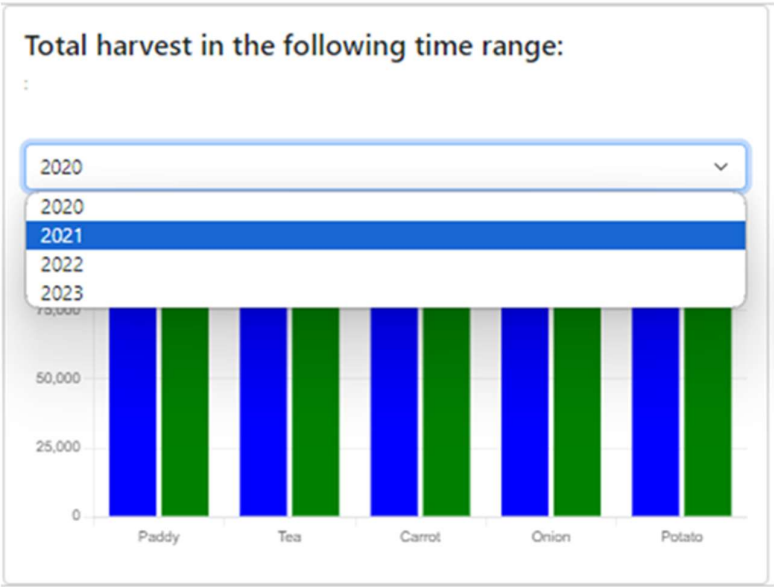



*Figure4.31: Preview of total harvest in selected timeframe*

```
<CCardBody>
        <CRow>
          <CCol>
            {/* Title displaying */}
            <h4>Total harvest in the following time range:</h4>
            <div className="small text-medium-emphasis">
              {' '}
              {/* Displaying start date and end date */}
              {formData.start_date} : {formData.end_date}
            </div>
            {/* selecting year */}
            <div style={{ height: 'auto', marginTop: '40px' }}>
              <CFormSelect custom name="year" id="year" onChange={handleYearChange}>
                <option value="2020">2020</option>
                <option value="2021">2021</option>
                <option value="2022">2022</option>
                <option value="2023">2023</option>
              </CFormSelect>
              {/* Barchart dispalying  */}
              <CChartBar
                style={{ height: '300px', marginTop: '10px' }}
                data={{
                  labels: harvestData.map((data) => data.crop_name),
                  // dataset for expected harvest amount
                  datasets: [
                    {
                      label: 'Expected Harvest Amount',
                      backgroundColor: 'blue',
                      data: harvestData.map((data) => data.total_estimated_harvested_amount),
                    },
                  // dataset for actual harvest amount
                    {
                      label: 'Actual Harvest Amount',
                      backgroundColor: 'green',
                      data: harvestData.map((data) => data.total_harvested_amount),}],}}
---//rest code//---
```

*Figure 4.30: Code snippet of display and interaction with harvest data in a specified time range in web application*

The code snippet in Figure 4.32, 'HarvestEstimatedVsActual' component is a functional React component responsible for fetching and displaying harvest data. It utilizes React's 'useState' hook to manage state variables such as harvestData, year, and formData. The 'useEffect' hook is employed to trigger a data-fetching function when the component mounts or when the selected year changes. This function, implemented using the axios library, fetches harvest data from the server based on the selected year and updates the harvestData state variable accordingly.

```
// Handle useState()
const HarvestEstimatedVsActual = () => {
  const [harvestData, setHarvestData] = useState([])
  const [year, setYear] = useState(new Date().getFullYear())
  const [formData, setFormData] = useState({ start_date: '', end_date: '' })

//Get dataset from database through an API call
  useEffect(() => {
    const getData = async () => {
      try {
        const response = await axios.get(`${API_BASE_URL}/report/harvest-amount-by-crop/${year}`)
        setHarvestData(response.data)
      } catch (error) {
        console.error('Error fetching data', error)
      }
    }

    getData()
  }, [year])
```

*Figure 4.32: Function of sending API call to get harvest data to specified route in REST API*

Code snippet in Figure 4.33 describes the REST API endpoint for the above-described scenario. This code defines a Flask route that handles GET requests for fetching aggregated data on harvested and estimated harvested amounts of crops for a specified agricultural year. It queries the database to aggregate this data and returns it as JSON, including crop IDs, total harvested amounts, estimated harvested amounts, and crop names.

```
@report_routes.route('/harvest-amount-by-crop/<int:agri_year>', methods=['GET'])
def get_harvest_amount_by_crop(agri_year):
    # Query the database for the total harvested amount and estimated harvested amount of each crop
    cultivation_infos = db.session.query(
        CultivationInfo.crop_id,
        func.sum(CultivationInfo.harvested_amount).label('total_harvested_amount'),
        func.sum (CultivationInfo.estimated_harvest).label('total_estimated_harvested_amount')
    ).filter(
        CultivationInfo.agri_year == agri_year
    ).group_by(
        CultivationInfo.crop_id
    ).all()

    # Convert the query results to a list of dictionaries
    harvest_amount_by_crop = [
        {'crop_id': info.crop_id, 'total_harvested_amount': info.total_harvested_amount,
         'total_estimated_harvested_amount': info.total_estimated_harvested_amount}
        for info in cultivation_infos
    ]

    # Get the crop names from the Crop table
    for data in harvest_amount_by_crop:
        crop = Crop.query.get(data['crop_id'])
        data['crop_name'] = crop.crop_name if crop else 'Unknown'

    return jsonify(harvest_amount_by_crop)
```

*Figure 4.33:Flask route that handles GET requests for fetching aggregated data on harvested and estimated harvested amounts of crops for a specified agricultural year*

### 2. Crop Yield Report

Code snippet in Figure 4.34(a), Figure 4.34(b) builds a React component called 'LankaMapByCropYieldAdmin' responsible for displaying a map and filtering data based on user selections (see Figure 4.35). The component renders a form allowing users to select various parameters such as crop type, year, month, province, district, and office. When it makes selections, the component sends requests to the backend API using Axios to retrieve data based on the chosen parameters. The received data is then used to update the map dynamically, on the map based on the total harvested crops amount in different districts. The map is rendered using the 'MapContainer' component from the 'react-leaflet' library (see Figure 4.36).

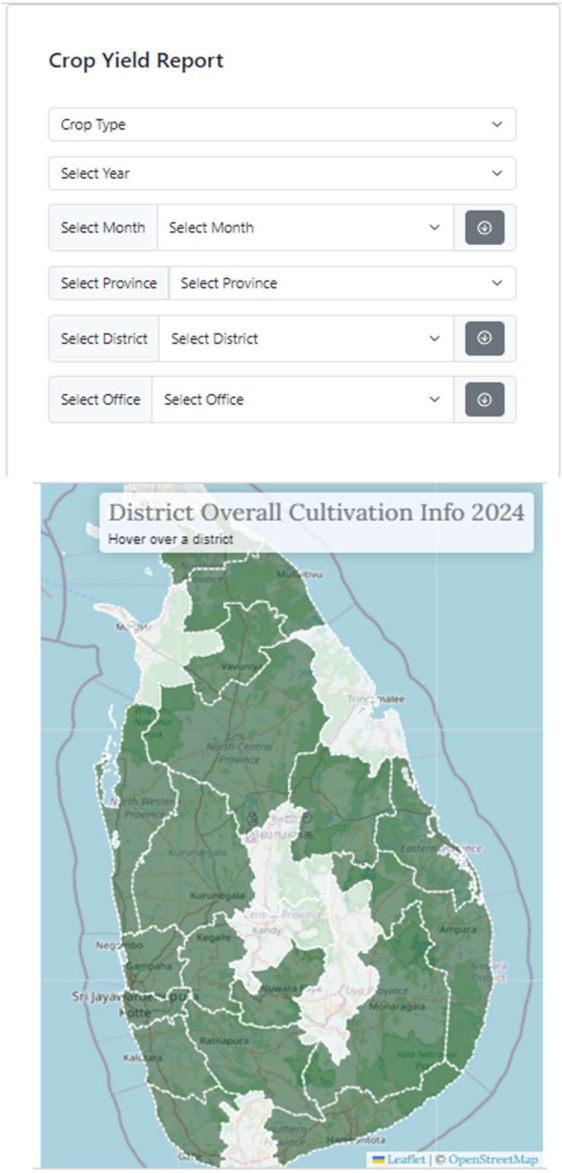



*Figure 4.35: displaying a map and filtering data based on user selections*

```
const LankaMapByCropYieldAdmin = () => {
  const center = [7.8731, 80.7718] // coordinates for Sri Lanka
  const [offices, setOffices] = useState([])
  const [crops, setCrops] = useState([])
  const monthNames = [//---months----//]
  const [formData, setFormData] = useState({year: '',month: '', crop_id: '',type:'',district: '',office_id: '',}
  const [districts, setDistrict] = useState([])
  const [filteredOffices, setFilteredOffices] = useState([])
  const [mapKey, setMapKey] = useState(0)
  const sri_lanka_provinces = [//----Provinces----//  ]

  //resets map in every form data ange
useEffect(() => {
    setMapKey((prevKey) => prevKey + 1)
  }, [formData])

//Fetch crop data on component mount
useEffect(() => {
    const token = localStorage.getItem('token')
    axios.get(`${API_BASE_URL}/crop/crops`, {
        headers: {
          Authorization: `Bearer ${token}`,},})
      .then((response) => {
        if (response.status === 200) {
          setCrops(response.data)}})
      .catch((error) => {
        console.log(error)})}, [])

const handleTypeSelect = (event) => {
    const { name, value } = event.target
    setFormData((prevFormData) => ({
      ...prevFormData,
      [name]: value,}))}

const handleYearSelect = async (event) => {--Handle the selection of year--}
const handleMonthSelect = async (event) => {--Handles the selection of month-}
const handleProvinceChange =async (event) =>{-Handles the province selection-}
const handleDistrictChange =async(event) =>{-Handles the district selection}
const filterOfficesByDistrict=(offices, district) =>{-Handles the office selection-}
```

API Call

*Figure 4.34 (a): Code snippet for rendering a map interface related to crop yield*

```
<MapContainer
                key={mapKey}
                center={center}
                zoom={8}
                style={{ height: '700px', width: '500px' }}
                dragging={false}
                touchZoom={false}
                doubleClickZoom={false}
                scrollWheelZoom={false}
                keyboard={false}
                zoomControl={false}
              >
                <TileLayer
                  url="https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png"
                  attribution='&copy; <a href="http://www.openstreetmap.org/copyright">OpenStreetMap</a>'
                  maxZoom={8}
                />
                <MapFeatureDataLayer formData={formData} />
              </MapContainer>
```

*Figure 4.36: Code snippet for setting up map container using mapContainer from react-leaflet library*

```
return (
    <CContainer fluid>
      <CCard>
        <CCardBody>
          <CRow>
            <CCol style={{ margin: '30px' }}>
              <h4>Crop Yield Report</h4>
              <div style={{ height: 'auto', marginTop: '40px' }}>
                <CInputGroup className={`mb-3`}>
                  <CFormSelect
                    custom
                    name="crop_id"
                    value={formData.crop_id}
                    onChange={handleTypeSelect}
                  >
                    <option value="">Crop Type</option>
                    {crops.map((crop, index) => (
                      <option key={index} value={crop.crop_id}>
                        {crop.crop_name}
                      </option>
                    ))}
                  </CFormSelect>
                </CInputGroup>
                <CInputGroup className={`mb-3`}>
                  <CFormSelect
                    custom
                    name="year"
                    value={formData.year}
                    onChange={handleYearSelect}
                  >
                    <option value="">Select Year</option>
                    <option value="2020">2020</option>
                    <option value="2021">2021</option>
                    <option value="2022">2022</option>
                    <option value="2023">2023</option>
                    <option value="2024">2024</option>

                -------//Thus the code continues the filters ; month, province, district, and office//--------
```

*Figure 4. 34(b): Code snippet for JSX structure for rendering a form interface for selecting various parameters related to crop yield reporting*

Code snippet in Figure 4.37 defines a Flask route to handle GET requests to the '/crops' endpoint. The route is protected by JWT authentication, ensuring that only authenticated users can access it. When a GET request is received, the function 'crops()' is executed. This function retrieves all crop records from the database using SQLAlchemy's query interface and serializes them into JSON format using a Marshmallow schema named 'crops_schema'. The serialized data is then returned as a JSON response to the client using Flask's 'jsonify()' function. In essence, this route provides authenticated users with access to a list of crop data stored in the database, facilitating interactions between the frontend and backend of the application.

```python
@crop_routes.route('/crops', methods=['GET'])
@jwt_required()
def crops():
    crop_list = Crop.query.all()
    result = crops_schema.dump(crop_list)
    return jsonify(result)
```

*Figure 4.37: Flask route for fetching crop records with authentication. (REST API)*

### 3. Field Mapping Report

Among the various types of reports supported by the system, the field mapping report is used to represent the geo location related data of the cultivations. According to the user input, Data is filtered and retrieved from the backend and it is shown in the map using React-leaflet. The following Figure 4.38 shows the frontend user interface of the report generating form. Users can download the data for relevant filters in CSV format by clicking the download icon as shown in the following.

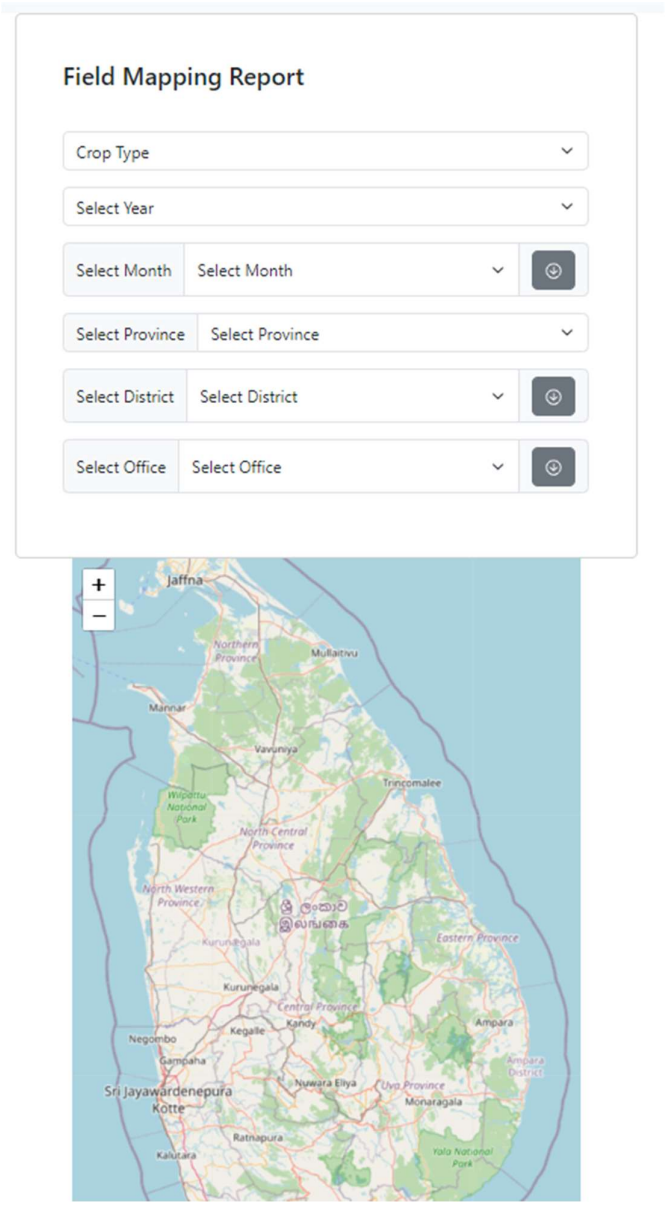



*Figure 4.38: frontend User interface of the Field map report form*

**Frontend User interface implementation**

This Figure 4.39 shows the form input components used to build the above UI form. When it is filled by the user, relevant dropdowns are loaded according to the previous user selection.

```
<CInputGroup className={`mb-3`}>
<CInputGroupText>Select Month</CInputGroupText>
<CFormSelect
  name="month"
  value={formData.month}
  onChange={handleMonthSelect}
>
  <option value="">Select Month</option>
  {monthNames.map((month, index) => (
    <option key={index} value={index + 1}>
      {month}
    </option>
  ))}
</CFormSelect>
<CInputGroupText>
  <CButton color="secondary">
    <CIcon icon={cilArrowCircleBottom} />
  </CButton>
</CInputGroupText>
/CInputGroup>
CInputGroup className={`mb-3`}>
<CInputGroupText>Select Province</CInputGroupText>
<CFormSelect
  onChange={handleProvinceChange}
  name="province"
  value={formData.province}
>
  <option value="">Select Province</option>
  {sri_lanka_provinces.map((province, index) => (
    <option key={index} value={province}>
      {province}
    </option>
  ))}
</CFormSelect>
```

```
</CInputGroup>
<CInputGroup className={`mb-3`}>
  <CInputGroupText>Select District</CInputGroupText>
  <CFormSelect
    name="District"
    value={formData.district}
    onChange={handleDistrictChange}
  >
    <option value="">Select District</option>
    {districts.map((district, index) => (
      <option key={index} value={district}>
        {district}
      </option>
    ))}
  </CFormSelect>
  <CInputGroupText>
    <CButton color="secondary">
      <CIcon icon={cilArrowCircleBottom} />
    </CButton>
  </CInputGroupText>
</CInputGroup>
<CInputGroup className={`mb-3`}>
  <CInputGroupText>Select Office</CInputGroupText>
  <CFormSelect
    name="office_id"
    value={formData.office_id}
    onChange={handleTypeSelect}
  >
    <option value="">Select Office</option>
    {filteredOffices.map((office, index) => (
      <option key={index} value={office.agri_office_id}>
        {office.name}
      </option>
    ))}
  </CFormSelect>
  <CInputGroupText>
    <CButton color="secondary">
      <CIcon icon={cilArrowCircleBottom} />
    </CButton>
  </CInputGroupText>
</CInputGroup>
```

*Figure 4.39 :Form of the field mapping reports with dynamic option update with the user inputs*

As shown in the following Figure 4.40, Crop types are retrieved from the database. The first time the component is rendered, data is retrieved by a REST API request. The react useEffect function is used to send the request and set the data in crop dropdown only once when the component is rendered.

```
  useEffect(() => {
// Retrieve the token from local storage (assumes token was previously saved)
const token = localStorage.getItem('token');

// Make a GET request to fetch crops data from the API
axios
  .get(`${API_BASE_URL}/crop/crops`, {
    headers: {
      // Include the Authorization header with the Bearer token for authentication
      Authorization: `Bearer ${token}`,
    },
  })
  .then((response) => {
    // Check if the response status is 200 (OK)
    if (response.status === 200) {
      // If successful, update the 'crops' state with the fetched data
      setCrops(response.data);
    }
  })
  .catch((error) => {
    // Log any errors that occur during the request
    console.log(error);
  });

}, []);  // Empty dependency array ensures this runs only once on component mount
```

*Figure 4.40: UseEffect to Run once when the component is rendered*

Once the crops are set to the drop down, users can select the crop type with other filters from the form. When the province is selected, the relevant districts are added to the dropdown using API request to the backend API as shown in the following Figure 4.41)

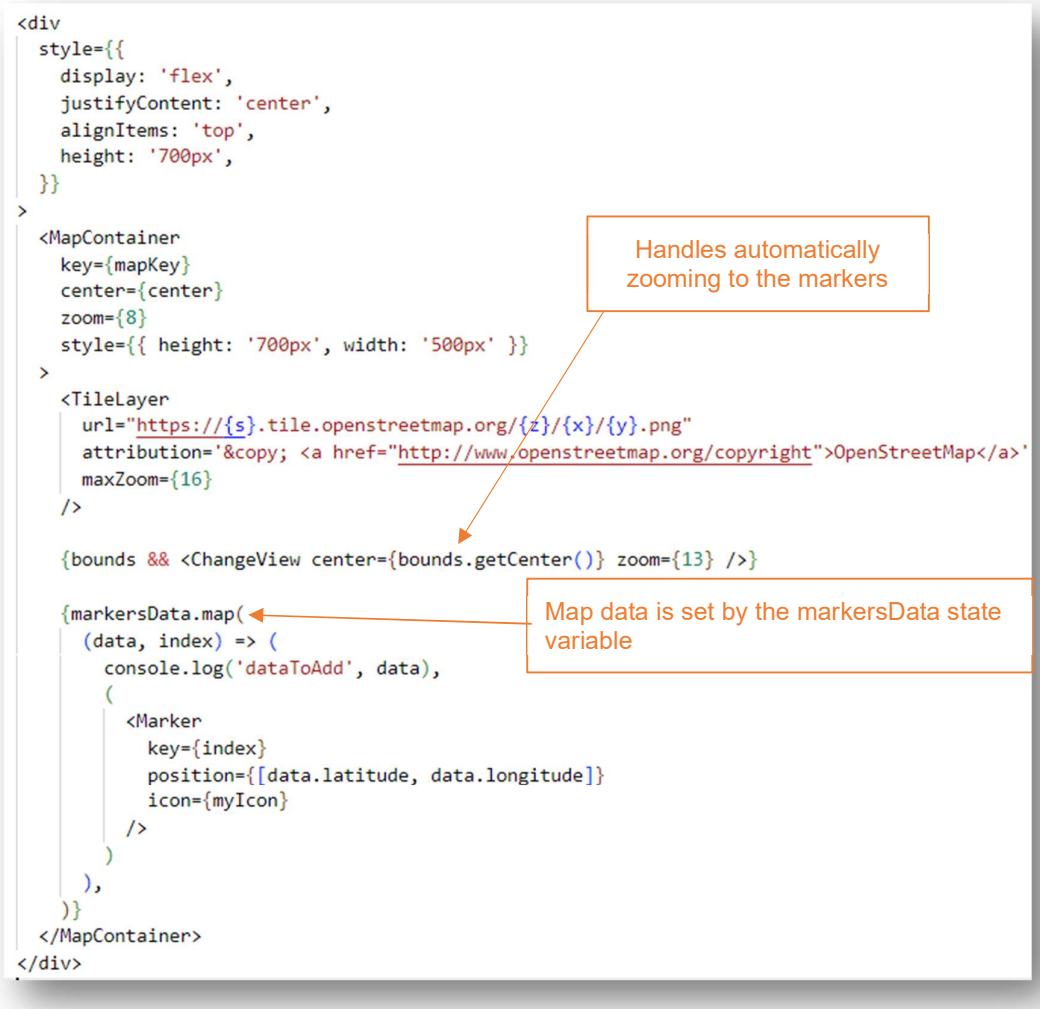
```
<CInputGroup className={`mb-3`}>
<CInputGroupText>Select District</CInputGroupText>
<CFormSelect
  name="District"
  value={formData.district}
  onChange={handleDistrictChange}
>
  <option value="">Select District</option>
  {districts.map((district, index) => (
    <option key={index} value={district}>
      {district}
    </option>
  ))}
</CFormSelect>
<CInputGroupText>
```

```
const [offices, setOffices] = useState([])
const [districts, setDistrict] = useState([])
const [filteredOffices, setFilteredOffices] = useState([])
```

```
// This function is called when the selected province changes.
const handleProvinceChange = async (event) => {
  const { name, value } = event.target; // Extract the name and value from the event.
  // Update the formData state with the new selected province.
  setFormData((prevFormData) => ({
    ...prevFormData,
    [name]: value, // Set the province value based on the user input.
  }));
  // Make an asynchronous request to get all offices and districts by the selected province.
  const response = await getAllOfficesAndDistrictsByProvince(value);
  // Update the offices and districts state with the data from the response.
  setOffices(response.data.offices);
  setDistrict(response.data.districts);
};
// This function is called when the selected district changes.
const handleDistrictChange = async (event) => {
  const { value } = event.target; // Extract the selected district value from the event.
  // Update the formData state with the new selected district.
  setFormData((prevFormData) => ({
    ...prevFormData,
    district: value, // Set the district value based on the user input.
  }));
  // Filter the offices by the selected district.
  const filteredOffices = filterOfficesByDistrict(offices, value);
  // Update the filteredOffices state with the filtered offices.
  setFilteredOffices(filteredOffices);
};
```

*Figure 4.41: Setting district and officers of the selected district to the state variables to be used in dropdowns*

Once the filters are added, users will receive data from the backend service by the function shown in Figure 4.42 and Figure 4.43. Then, the markers will be added to the map, and the map is zoomed automatically according to the filters given. The code related to this functionality is as follows.




```
<div
  style={{
    display: 'flex',
    justifyContent: 'center',
    alignItems: 'top',
    height: '700px',
  }}
>
  <MapContainer
    key={mapKey}
    center={center}
    zoom={8}
    style={{ height: '700px', width: '500px' }}
  >
    <TileLayer
      url="https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png"
      attribution='&copy; <a href="http://www.openstreetmap.org/copyright">OpenStreetMap</a>'
      maxZoom={16}
    />

    {bounds && <ChangeView center={bounds.getCenter()} zoom={13} />}

    {markersData.map(
      (data, index) => (
        console.log('dataToAdd', data),
        (
          <Marker
            key={index}
            position={[data.latitude, data.longitude]}
            icon={myIcon}
          />
        )
      ),
    )}
  </MapContainer>
</div>
```

*Figure 4.42: Code snippet for Setting marker data to the map*

```
            // The function passed to useEffect will run after the render is committed to the screen.
useEffect(() => {
  // Check if all necessary fields in formData are filled
  if (
    formData.year !== '' && // Ensure the 'year' field is filled
    formData.crop_id !== '' && // Ensure the 'crop_id' field is filled
    formData.month !== '' && // Ensure the 'month' field is filled
    formData.district !== '' && // Ensure the 'district' field is filled
    formData.office_id !== '' // Ensure the 'office_id' field is filled
  ) {
    // If all fields are filled, make a request to get the cultivation map info
    searchCultivationMapInfoByDistrictMonthlyOffice(
      formData.year,
      formData.crop_id,
      formData.month,
      formData.district,
      formData.office_id,
    )
    .then((response) => {
      // If the request is successful (status code 200), update the markersData state
      if (response.status === 200) {
        setMarkersData(response.data); // Update the markersData with the response data
        console.log('response', response.data); // Log the response for debugging
      }
    })
    .catch((error) => {
      // If the request fails, do nothing
      console.error(error); // Log the error for debugging purposes
    });
  }
}, [
  // The array of dependencies. When one of these values changes, the function will be run again.
  formData.year, formData.crop_id, formData.month, formData.district, formData.office_id
```

*Figure4.43: Code snippet of UseEffect hooks to retrieve data once input are filled, using API service function "searchCultivationMapInfoByDistrictMonthlyOffice*

Once the valid filters are provided, the map will be loaded as in the following Figure 4.44.

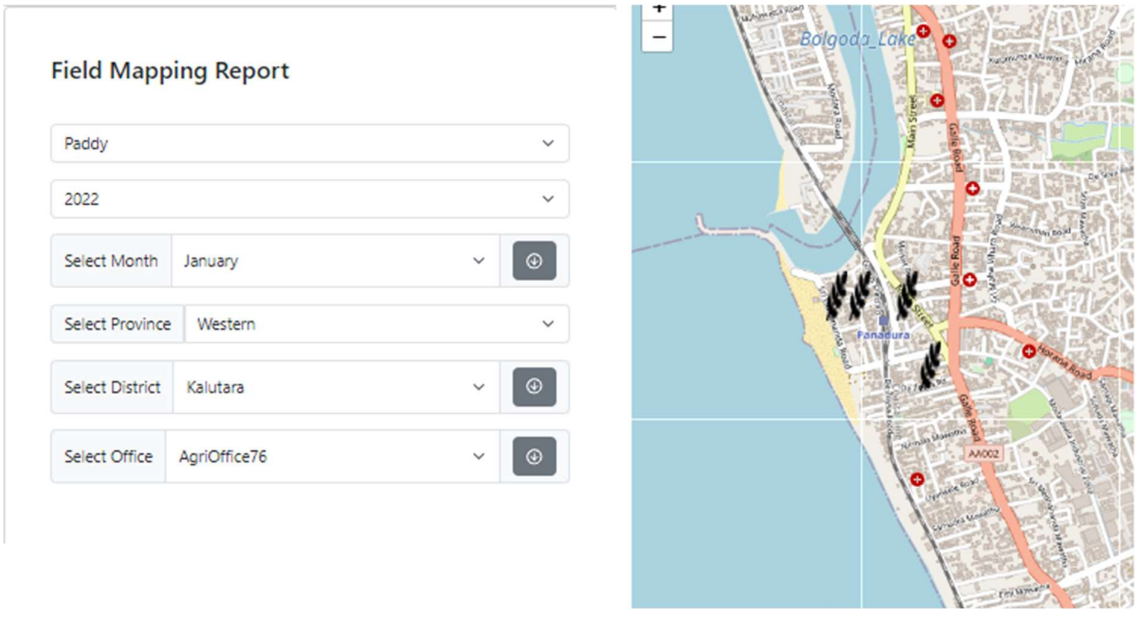



*Figure 4.44: Markers shown in the map*

**Backend service function for field report**

There are several functions used to filter the data and send the response to the request sender as shown in below Figure 4.45.

```
@report_routes.route('/search/cultivation-map/monthly/district/office', methods=['POST'])
def search_crop_cultivation_map_by_monthly_district_office():
    # Get the request data
    data =  request.get_json()

    # Extract the required parameters from the data
    agri_year = data.get('year')
    month = data.get('month')
    crop_id = data.get('crop_id')
    district = data.get('district')
    office_id = data.get('office_id')

    # Query the database for the cultivation information
    result = db.session.query(
        AgriOffice.district,
        Crop.crop_name,
        CultivationInfo.cultivation_info_id,
        CultivationInfo.longitude,
        CultivationInfo.latitude,
        AgriOffice.agri_office_id,
    ).join(
        Farm, Farm.farm_id == CultivationInfo.farm_id  # Join with Farm table
    ).join(
        Farmer, Farmer.user_id == Farm.farmer_id  # Join with Farmer table
    ).join(
        AgriOffice, AgriOffice.agri_office_id == Farm.office_id  # Join with AgriOffice table
    ).join(
        Crop, Crop.crop_id == CultivationInfo.crop_id  # Join with Crop table
    ).filter(
        # Apply filters based on the request parameters
        CultivationInfo.agri_year == agri_year,
        CultivationInfo.crop_id == crop_id,
        extract('month', CultivationInfo.estimated_harvesting_date) == month,
        AgriOffice.district == district,
        AgriOffice.agri_office_id == office_id
    ).all()

    # Return the result as a JSON response
    return jsonify([row._asdict() for row in result])
```

*Figure 4.45: Backend service function*

### 4.6.4 Message Broadcasting

One of the main use cases of the requirement was to introduce a feature to directly send messages to agricultural field officers and to the farmers. System allows admins and officers to send and keep information on sent messages. The Figure 4.46 explains the implementation of sending emails to system administrative role officers to agricultural field officers.

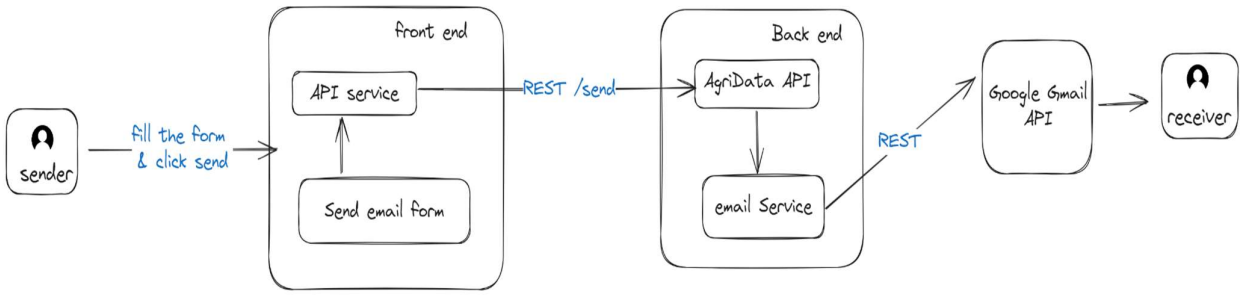



*Figure 4.46: Email sending feature in "Ceylon AgriData " system*

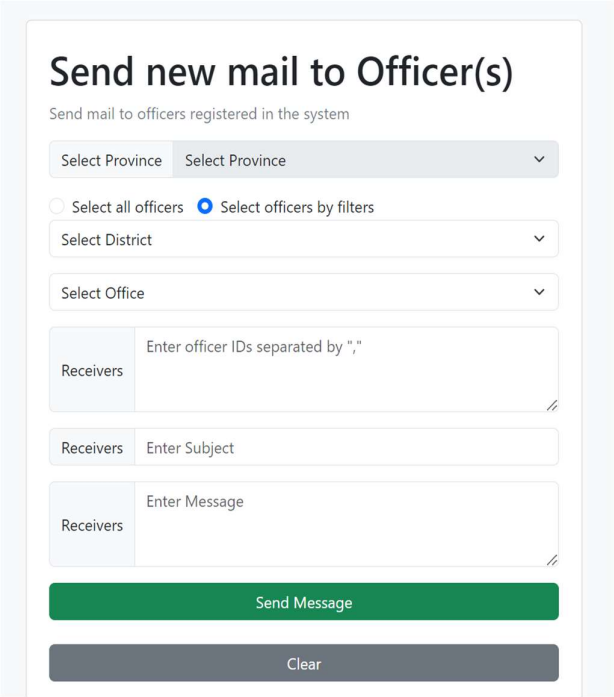



*Figure 4.47: UI of broadcasting emails feature*

Sender gets two options to send messages to all officers or sending officers by given filters. This form in Figure 4.47, is generated by React form as shown above. It uses react core-ui radio buttons to get the user input and the selection is set to declared state variables using React's

'useState' hook in the functional component. According to the state variable value, the input elements are disabled as shown in Figure 4.48.


*Figure 4.48: Selection of recipients*

When the user selects “Select all officers option” then, the user needs to select the province that all officers should select. Once the province is selected and the subject and the message is added, the user can click on the send message that calls an API service function to send the request to the backend API along with the user inputs. Related code snippet is in Figure 4.49.


```
<div className="d-grid">
  {isLoading ? (
    <Spinner animation="border" variant="primary" />
  ) : (
    <CButton
      color="success"
      onClick={handleNewItemSendButtonSubmit}
      disabled={isFormEmpty}
    >
      Send Message
    </CButton>
  )}
  <br />
  <CButton color="secondary" onClick={handleSendMessageCleanForm}>
    Clear
  </CButton>
</div>
```

```
const handleNewItemSendButtonSubmit = async (event) => {
event.preventDefault()
console.log(messageSendingForm)
try {
  if (
    messageSendingForm.isAllSelected === true &&
    messageSendingForm.message !== '' &&
    messageSendingForm.subject !== '' &&
    messageSendingForm.province !== ''
  ) {
    setIsLoading(true)
    const response = await sendBulkMailsByProvice(
      messageSendingForm.province,
      messageSendingForm.subject,
      messageSendingForm.message,
    )
    console.log(response)
    if (response.status === 200) {
      alert('Messages were dispatched Successfully!')
    } else if (response.status === 409) {
      alert(response.data.message)
    } else {
      alert('Message sending to the system failed. Contact API service in charge.')
    }
  } else if (
    messageSendingForm.isAllSelected === false &&
    messageSendingForm.message !== '' &&
    messageSendingForm.subject !== '' &&
    messageSendingForm.officerList.length !== 0
  ) {
    setIsLoading(true)
    const response = await sendMailToOfficer(
      messageSendingForm.subject,
      messageSendingForm.message,
      messageSendingForm.officerList,
    )
```

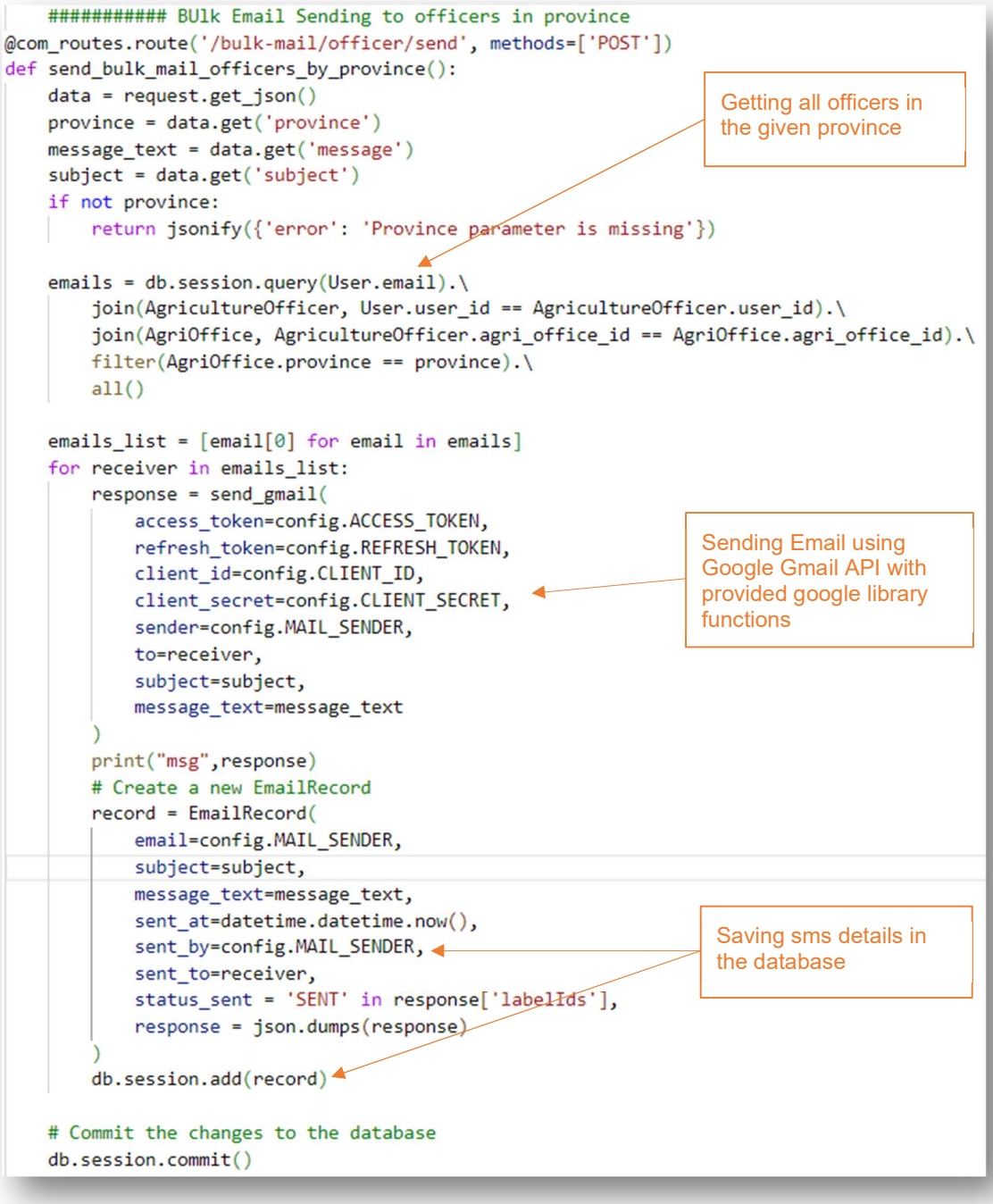
*Figure 4.49: API service function*

The following code snippet in Figure 4.50 is the API function calls that send the request to Backend API with the details of the email message.


```
import axios from 'axios'
import { API_BASE_URL } from 'src/Config'
const sendBulkMailsByProvice = async (provinceName, subject, message) => {
  const token = localStorage.getItem('token')
  try {
    const response = await axios.post(
      `${API_BASE_URL}/communication/bulk-mail/officer/send`,
      { province: provinceName, subject: subject, message: message },
      {
        headers: {
          Authorization: `Bearer ${token}`,
        },},)
    return response
  } catch (error) {
    console.error('Failed to fetch farm details:', error)
    return}}
const sendMailToOfficer = async (subject, message, officerList) => {
  const token = localStorage.getItem('token')
  try {
    const response = await axios.post(
      `${API_BASE_URL}/communication/mail/officer/send`,
      { receivers: officerList, subject: subject, message: message },
      {
        headers: {
          Authorization: `Bearer ${token}`,
        },},)
    return response
  } catch (error) {
    console.error('Failed to fetch farm details:', error)
    return}}
```

*Figure 4.50: API calls that sends requests to backend API*

The following code snippet Figure 4.51 shows the function that is called by the front-end application as explained above. First, it gets all the officers in the given province, then it iterates the email list using a for loop and it sends the emails and finally saves the response with relevant information of each message in the database. If something goes wrong with dispatching emails to Google, then those details can be checked using the email table from the database, as it saves the response, ensuring better message sending functionality to the system.




```python
    ########### BUlk Email Sending to officers in province
@com_routes.route('/bulk-mail/officer/send', methods=['POST'])
def send_bulk_mail_officers_by_province():
    data = request.get_json()
    province = data.get('province')
    message_text = data.get('message')
    subject = data.get('subject')
    if not province:
        return jsonify({'error': 'Province parameter is missing'})

    emails = db.session.query(User.email).\
        join(AgricultureOfficer, User.user_id == AgricultureOfficer.user_id).\
        join(AgriOffice, AgricultureOfficer.agri_office_id == AgriOffice.agri_office_id).\
        filter(AgriOffice.province == province).\
        all()

    emails_list = [email[0] for email in emails]
    for receiver in emails_list:
        response = send_gmail(
            access_token=config.ACCESS_TOKEN,
            refresh_token=config.REFRESH_TOKEN,
            client_id=config.CLIENT_ID,
            client_secret=config.CLIENT_SECRET,
            sender=config.MAIL_SENDER,
            to=receiver,
            subject=subject,
            message_text=message_text
        )
        print("msg",response)
        # Create a new EmailRecord
        record = EmailRecord(
            email=config.MAIL_SENDER,
            subject=subject,
            message_text=message_text,
            sent_at=datetime.datetime.now(),
            sent_by=config.MAIL_SENDER,
            sent_to=receiver,
            status_sent = 'SENT' in response['labelIds'],
            response = json.dumps(response)
        )
        db.session.add(record)

    # Commit the changes to the database
    db.session.commit()
```

*Figure 4.51: Backend API function that is called by frontend*

For Google API call authentication, the service is needed to set the *client id, client secret and access token, refresh token* as configurations in the initialization of the system. Here, the authorization grant type of *OAuth2.0* is used as recommended by the google. These tokens can be generated using the following link for google support docs.

Link:https://developers.google.com/gmail/api/auth/scopes#configure_oauth_20_for_authorization

## 4.7 Summary

Chapter 4 explores the technical aspects of implementing the "Ceylon AgriData" system, which aims to modernize data collection in Sri Lanka's agricultural sector. The chapter details the adoption of a service-oriented architecture, employing Flutter and React for the frontend, Python Flask REST API for the backend, and MySQL for data storage. The implementation ensures efficiency, accuracy, and scalability. Third party libraries and services such as Leaflet.js and Google APIs enhance functionality. Detailed explanations of code components cover user authentication, routing, HTTP request handling, and business logic implementation, highlighting best practices in software development. The integration of security measures like JWT authentication and CORS configuration underscores the system's reliability. Overall, the chapter provides a comprehensive guide to the technical implementation of "Ceylon AgriData," showcasing its innovative approach to agricultural data management.

# Chapter 5 – Testing and Evaluation

## 5.1 Introduction

The ‘Ceylon AgriData’ platform is designed to enhance the efficiency and productivity of the agricultural sector. It integrates various stakeholders, including administrators, agriculture officers, farmers, vendors, and researchers, facilitating a seamless exchange of information and resources. This chapter provides an overview of the rigorous testing process undertaken to determine the reliability, functionality, and user-friendliness of the ‘Ceylon AgriData’ system.

## 5.2 Related Testing Types Utilized

In the development of the "Ceylon AGriData" system, various testing types were employed to ensure its robustness and reliability. These included *unit testing*, *integration testing*, *end-to-end testing*, *cross-browser testing*, and *user acceptance testing*. Unit testing focused on verifying individual units or components. Integration testing validated the interaction between different modules for testing the backend API's connection with the database. End-to-end testing ensured the seamless flow of both mobile and web applications to validate user interface and backend functionality. Cross-browser testing ensured compatibility across different browsers for the React web application. Finally, user acceptance testing, assessed the system's compliance with user requirements. These testing types collectively ensured the quality and effectiveness of the "Ceylon AGriData" system across its various components and functionalities

## 5.3 Testing Methodology

The testing methodology for the "Ceylon AgriData" system was structured to ensure thorough validation of its mobile application, web application, and backend API. Beginning with the identification of test tasks and definition of corresponding test cases, the testing process encompassed various phases. Unit testing involved both manual and automated approaches, with automated tests specifically targeting user services. Integration testing verified the interaction between components, utilizing Postman to test the backend API's connection with the database. First we use locally hosted applications for testing, However, for facilitating end-to-end testing of the mobile app, Ngrok, an open-source tool, was employed to establish a secure tunnel between the mobile application and the backend API, which was running locally on localhost 5000. This setup enabled the execution of API calls from the mobile application

for comprehensive validation. End-to-end testing ensured the seamless flow of both web and mobile applications, conducted manually to validate user interface functionality and backend processes. Next, Same approach was used to do the end-to-end tests of the web application. Cross-browser testing was performed on the React web application to guarantee compatibility across different browsers. Exploratory testing was employed to systematically explore the application for unforeseen issues, while user acceptance testing, facilitated by a questionnaire, assessed compliance with user requirements. Test execution predominantly occurred manually, with comprehensive documentation of test cases and results provided in tables for analysis.

## 5.3 Testing of Mobile Application

### 5.3.1 Unit testing – Mobile Application

In the Unit Testing phase, some major test cases specified in Table 5.1 were rigorously examined using the Android Studio emulator, tested manually. The primary focus during this phase was on ensuring the accurate construction of widgets and the proper functioning of associated functionalities within the Android application. By utilizing the Android Studio emulator, it could assess the behavior of individual components and functions within the application's codebase. This approach enabled thorough validation of the application's core building blocks, ensuring their correctness and reliability before proceeding to subsequent testing phases. Through Unit Testing, potential issues related to widget construction and function execution were systematically identified and addressed.

*Table 5.1:Test cases used in unit testing of mobile application*

| Test Case Id | Test Case | Expected Result | Actual Result |
|---|---|---|---|
| 1 | Splash page loading | Splash page loading successfully | Splash page loaded |
| 2 | Navigates to login page | Successfully navigates to login page automatically after splash page | Login page loaded automatically after splash page |
| 3 | Building login page widgets | Successfully built the login page | All widgets were displayed |

| | | | |
|---|---|---|---|
| 4 | Login function | Login function working well with button click | Login function worked |
| 5 | Register agriculture officer function | Register function working well with button click | Register function worked |
| 6 | Building Register page widgets | Successfully built the register page widgets | All widgets were displayed |
| 7 | Build farmer manager page | Successfully built the page and preview all buttons | Page was loaded and buttons were previewed |
| 8 | Build farm manager page | Successfully built the page and preview all buttons | Page was loaded and buttons were previewed |
| 9 | Build cultivation manager page | Successfully built the page and preview all buttons | Page was loaded and buttons were previewed |
| 10 | Build aid manager page | Successfully built the page and preview all buttons | Page was loaded and buttons were previewed |
| 11 | Build disaster manager page | Successfully built the page and preview all buttons | Page was loaded and buttons were previewed |
| 12 | Input farmer details | Working all textformfields | Was able to insert details through textformfields |
| 13 | Input farm details | Working all textformfields | Was able to insert details through textformfields |
| 14 | Input cultivation details | Working all textformfields | Was able to insert details through textformfields |
| 15 | Input aid details | Working all textformfields | Was able to insert details through textformfields |
| 16 | Input disaster details | Working all textformfields | Was able to insert details through textformfields |
| 17 | Functioning dropdown in add cultivation page | Successfully working dropdown | Dropdown was not working well |
| 18 | Functioning dropdown in add disaster page | Successfully working dropdown | Dropdown was working well |
| 19 | Register farmer function | Register function working well with button click | Register function worked |

| | | | |
|---|---|---|---|
| 20 | Add cultivation function | Add cultivation function working well with button click | Add cultivation function worked |
| 21 | Add aid function | Add aid function working well with button click | Add aid function worked |
| 22 | Get location button functioning | Successfully functioning get locations | Current locations were fetched successfully |
| 23 | Logout function | Logout function working well with button click | Logout function worked |
| 24 | Building Main menu | Building successfully | Main menu successfully previewed but the logo was not previewed |
| 25 | Search function | Successfully triggers when button click | Searched and previewed results |
| 26 | Update/Delete function | Successfully triggers when button-search and update or delete clicked | Successfully listed search result and could edit and update it , Finally could deleted the result. |
| 27 | Textformfields of Update/Delete | Successfully Working all textformfields | Was able to update details through textformfields |
| 28 | Home icon | When presses the icon, successfully navigates to home page | Successfully navigated to home page |
| 29 | Get current location | When press the button, navigates to get location page and fetch current location successfully | Successfully fetched the location (longitude and latitude) |

Overall, the unit testing results indicate a high level of success in building widgets, implementing functions, and achieving expected behaviors within the Android application. The majority of test cases aligned with their expected results, demonstrating effective construction of pages, successful execution of functions such as login, registration, and data input, as well as proper navigation and interaction with various elements. However, some discrepancies were

noted, such as the malfunctioning dropdown in the add cultivation page and the absence of the logo preview in the main menu. These issues highlight areas for improvement to ensure full functionality and consistency across all aspects of the application.

### 5.3.2 Exploratory Testing – Mobile Application

Exploratory testing was manually performed on the mobile application by navigating through its pages and verifying the correct functioning of the mobile phone's back button for navigation. The results indicated successful navigation, and it was confirmed that the application smoothly facilitated movement between pages.

### 5.3.3 Integration testing – Mobile Application

Integration testing was conducted to verify the seamless connection between the mobile application, REST API, and database. As outlined in the methodology, Ngrok was utilized to facilitate this process as previously mentioned. Major test cases listed in Table 5.2 were executed manually, and the results were captured for analysis.

*Table 5.2: Test cases for integration testing in mobile application and captured results*

| **Id** | Description | Pre-condition | Steps | Expected Results | Actual Results |
|---|---|---|---|---|---|
| 1 | Valiate login user with correct input | Need stable internet connection, User should be on Login page, User should be a agriculture field officer | Enter valid email and password | User logged successfully | User logged into system |
| 2 | Validate login user with incorrect input | Need stable internet connection, User should be on Login | Enter either invalid email or password | Preview dialog box saying login failed | Previewed the message |

| | | | | | |
|---|---|---|---|---|---|
| | | page, User should be a agriculture field officer | | | |
| 3 | Validate login user with incorrect role id | Need stable internet connection, User should be on Login page, User should be a agriculture field officer | Enter correct username or password, but user's role not be equal t 4 | Preview dialog box saying invalid login | Previewed the message |
| 4 | Register a farmer | Need stable internet connection, User should be logged in | Enter all details correctly, Click submit button | Preview toast message saying registered successfully | Previewed the message |
| 5 | Register a farm | Need stable internet connection, User should be logged in | Enter all details correctly, Click submit button | Preview toast message saying registered successfully | Previewed the message |
| 6 | Add a cultivation | Need stable internet connection, User should be logged in | Enter all details correctly, Click submit button | Preview toast message saying registered successfully | Previewed the message |

| | | | | | |
|---|---|---|---|---|---|
| 7 | Add an aid distribution | Need stable internet connection, User should be logged in | Enter all details correctly, Click submit button | Preview toast message saying registered successfully | Previewed the message |
| 8 | Add a disaster record | Need stable internet connection, User should be logged in | Enter all details correctly, Click submit button | Preview toast message saying registered successfully | Previewed the message |
| 9 | Search Farmers | Need stable internet connection, User should be logged in, Should available farmer records | Enter user id or TaxFileNo or Office Id or Field Area Id, Click search button | Preview search results | Previewed results |
| 10 | Search Farms | Need stable internet connection, User should be logged in, Should available farm records | Enter farm Id or farmName or Owner NIC or Type, Click search button | Preview search results | Previewed results |

| | | | | | |
|---|---|---|---|---|---|
| 11 | Search Cultivation | Need stable internet connection,<br><br>User should be logged in, Should available cultivation records | Enter farm Id or farmName or Owner NIC or Type,<br>Click search button | Preview search results | Previewed results |
| 11 | Search aid distribution records | Need stable internet connection, User should be logged in, Should available records | Enter aid Id or farmer id or AgriOfficeId or aidType or In_charged Agri_officer_Id, Click search button | Preview search results | Previewed results |
| 12 | Search disaster records | Need stable internet connection, User should be logged in, Should available disaster records | Enter type or farm_id or farmer_id or Type,<br>Click search button | Preview search results | Previewed results |

| | | | | | |
|---|---|---|---|---|---|
| 13 | Update farmer records | Need stable internet connection, User should be logged in, Should available particular record | Enter userId, Search for results, Update record, Click Update button | Previewed searched record and preview toast message of updated successfully | Previewed message |
| 14 | Update farmer records | Need stable internet connection, User should be logged in, Should available particular record | Enter userId, Search for results, Update record, Click Update button | Previewed searched record and preview toast message of updated successfully | Previewed message |
| 15 | Update farm records | Need stable internet connection, User should be logged in, Should available particular record | Enter farmId, Search for results, Update record, Click Update button | Previewed searched record and preview toast message of updated successfully | Previewed message |
| 16 | Update cultivation records | Need stable internet connection, | Enter cultivationId, | Previewed searched record and | Previewed message |

| | | | | | |
|---|---|---|---|---|---|
| | | User should be logged in, Should available particular record | Search for results, Update record, Click Update button | preview toast message of updated successfully | |
| 17 | Update aid records | Need stable internet connection, User should be logged in, Should available particular record | Enter aidId, Search for results, Update record, Click Update button | Previewed searched record and preview toast message of updated successfully | Previewed message |
| 18 | Update disaster records | Need stable internet connection, User should be logged in, Should available particular record | Enter disasterId, Search for results, Update record, Click Update button | Previewed searched record and preview toast message of updated successfully | Previewed message |
| 19 | Delete farmer records | Need stable internet connection, User should be logged in, Should | Enter userId, Search for results, Click delete button | Previewed searched record and preview toast message of | Previewed message |

| | | | | | |
|---|---|---|---|---|---|
| | | available particular record | | deleted successfully | |
| 20 | Delete farm records | Need stable internet connection, User should be logged in, Should available particular record | Enter farmId, Search for results, Click delete button | Previewed searched record and preview toast message of deleted successfully | Previewed message |
| 21 | Delete cultivation records | Need stable internet connection, User should be logged in, Should available particular record | Enter cultivationId, Search for results, Click delete button | Previewed searched record and preview toast message of deleted successfully | Previewed message |
| 22 | Delete aid records | Need stable internet connection, User should be logged in, Should available particular record | Enter aidId, Search for results, Click delete button | Previewed searched record and preview toast message of deleted successfully | Previewed message |

| | | | | | |
|---|---|---|---|---|---|
| 23 | Delete disaster records | Need stable internet connection, User should be logged in, Should available particular record | Enter disasterId, Search for results, Click delete button | Previewed searched record and preview toast message of deleted successfully | Previewed message |
| 24 | Broadcast messages | Need stable internet connection, User should be logged in, Should available farmer mail addresses | Get recipient list, Enter Subject, Enter message body, Click send message button | Preview successfully sent toast message | Previewed message successfully |
| 25 | Logout | Need internet connection, Navigate to main menu | Click Logout in main menu | Navigates to login page successfully | Navigated to login page |

All test cases were successfully passed, indicating that the mobile application effectively triggered API calls to the backend API and accessed the database.

The "Ceylon AgriData" mobile app was smoothly installed and operated on Samsung A30 running Android 11, and Oppo F19 pro, running Android 13 showcasing its compatibility and reliability on the latest Android version and device model. This underscores the importance of

thorough device compatibility testing in ensuring optimal performance and user experience across diverse platforms.

## 5.4 Testing of Back-end Services, REST APIs

### 5.4.1 Unit Testing – Backend Services & APIs

The backend testing phase of the 'Ceylon AgriData' platform utilized Python's built-in module, 'unittest,' to execute unit tests across diverse scenarios. 'unittest' draws inspiration from the xUnit architecture, a widely adopted framework for unit testing. The objective was to verify the functionality of each component within the 'Ceylon AgriData' platform.

Creating a test environment for unit testing with Python's `unittest` module involves organizing the project structure with dedicated directories for source code and tests. Within the test's directory, Python files containing test cases are written, importing relevant modules and defining test methods. Tests are executed using the 'python -m unittest path for test file' command, with results indicating the success or failure of each test case, aiding in debugging and ensuring code reliability. The related test cases, input data, test steps, expected results and test status for 'User' entity are summarized in Table 5.3. See the unittest sample code in *Appendix H*.

*Table 5.3:Test cases for user related functionalities in unit test of API*

| Test Case Name | Input Data | Test Steps | Expected Results | Actual Results |
|---|---|---|---|---|
| User Registration | User Details: firstName, MiddleName, lastName, NIC, Email, Password, DOB, Role | 1. Input relevant data into the system. And commit the data to the database.<br>2. Assert result | Registration succeeds | Registration succeeded |

| | | | | |
|---|---|---|---|---|
| User Login | Email<br>Password | 1. Verify that the email and password provided correspond with the entered input data.<br>2. Assert result | Login succeeds | Login succeeded |
| User Deletion | User Id | 1. Check if a user exists in the system with the provided email address.<br>2. If a user is found, delete the user account.<br>3. Assert result | Deletion succeeds | Deletion succeeded |
| User update | Data Need to get updated (User details) and user_id | 1. Check if a user exists in the system with the provided user_id.<br>2. If a user is found, update the user account with given input data<br>3. Assert result | Update succeed | Update succeeded |
| Search User | Filters need to filter the result | 1. Specify filters to search for users with a specific first name and define pagination parameters<br>2. Perform the search operation using the defined filters.<br>3. Assert result | Search succeed | Search succeeded |
| Validate User | Use_id<br>Email | 1. Invoke the function with predefined parameters (user_id and email)<br>2. Assert the result | Validate succeed | Validate succeeded |
| Get User Information by Id | user_id | 1. Invoke the function with predefined parameters (user_id)<br>2. Retrieve user information<br>3. Assert the result | Successfully retrieving user information | Successfully retrieved user information |
| Get User Information by Email | Email | 1. Invoke the function with predefined parameters (Email)<br>2. Retrieve user information | Successfully retrieving user information | Successfully retrieved user information |

| | | | | |
|---|---|---|---|---|
| | | 3. Assert the result | | |
| Get user access token | Email | 1. Retrieves the user with the specified email address from the database, if it exists<br>2. the test proceeds to generate an access token for the user<br>3. Asserts that the generated token is not None | Successfully get the access token | Successfully got the access token |

**Unit testing code explanation**

Following Figure 5.1 showcases the code snippet for unit test conducted by 'Python unittest', for user registration functionality. The unit test method assesses the user registration process by simulating the creation of a test user and invoking the registration function. It confirms the success of registration based on the returned values, validating the system's ability to register users effectively and provide the expected success message. The 'test_user_registration' test method begins by accessing the application context to interact with the Flask application environment. A test user object is then generated with simulated user data, encompassing various fields such as first name, last name, NIC (National Identification Card), email, password, date of birth, and role. Subsequently, the 'register_user' function is called with the test user object, facilitating the user registration process. The function returns a tuple comprising a Boolean value indicating registration success (isSuccess) and a descriptive message. Assertions are utilized to validate the registration process: self.assertTrue(isSuccess) verifies successful registration, while self.assertEqual(message, 'Registration success!') ensures that the returned message aligns with the expected message for successful registration.

```python
def test_user_registration(self):
    with self.app.app_context():
        user = User(
            first_name='fName',
            middle_name='mName',
            last_name='lName',
            nic='testNIC',
            email='test@example.com',
            password='test',
            dob=parse_date('1990-01-01'),
            role=1,
        )
        isSuccess, message = register_user(user)
        self.assertTrue(isSuccess)
        self.assertEqual(message, 'Registration success!')
```

*Figure 5.1: Code snippet of user registration unit test function*

### 5.4.1 Integration testing - Backend Services & APIs

Integration testing was employed manually by connecting the system using Postman, focusing on verifying the functionality of backend API endpoints while ensuring authorization and SQLAlchemy connections. Postman served as a comprehensive platform for executing HTTP REST API requests to the server, allowing for thorough testing of API endpoints. The testing process involved validating the behavior of each endpoint, including the required authorization mechanisms, to ensure secure access. Additionally, As the backend APIs and the locally hosted MYSQL database was deployed, Postman facilitated the testing of Backend APIs, SQLAlchemy functionalities with the database, ensuring seamless interaction with the database. This integrated approach ensured that the API functions correctly within the broader system context, covering both functional and non-functional aspects of the application.

The testing process involved defining API endpoints and request methods, organizing them into a Postman collection. Testing was performed to verify the functionality of each API endpoint, including proper authentication mechanisms to system backend using JWT bearer token as the Authorization header. This provides valuable insights into the security and reliability of the backend API functionality.

'User' entity-related endpoints in the backend API, includes endpoints for user registration, login, retrieval of user information, updating user details, and deleting user accounts and test results are summarized in Table 5.4. Integration testing ensures that these endpoints function

correctly in a real-world scenario, handling various HTTP requests and responses appropriately. Additionally, authentication and authorization mechanisms are thoroughly tested to ensure that user access is securely managed. The testing process also involves verifying the integration of the SQLAlchemy ORM with the database backend to ensure seamless data operations. Through comprehensive integration testing, the 'User' entity endpoints are evaluated to ensure reliability, security, and adherence to specified requirements. Figures 5.2, 5.3 and 5.4 provide a visual representation of the testing conducted in Postman as supporting evidence.

*Table 5.4:Test Cases for User related functionalities of integration testing of API*

| **Test Case** | **End Points** | **Input Data** | **Authorization** | **Expected Result** | **Status** |
|---|---|---|---|---|---|
| User Registration | ‘/user/register’ | User Details: firstName, MiddleName, lastName, NIC, Email, Password, DOB, Role | None | Registration succeeds | Registration succeeded |
| User Login | ‘/user/login’ | Email Password | None | Login succeeds | Successfully LoggedIn |
| User Deletion | '/user/<int:userid>' | User Id | Bearer Token | Deletion succeeds | Successfully Deleted |
| User update | ‘/user/update/’ | Data Need to get updated (User details) and user_id | Bearer Token | Update succeeds | Updated successfully |
| Search User | ‘/user/search’ | Filters need to filter the result | Bearer Token | Search succeeds | Previewed search results |
| Validate User | ‘/user/validate’ | Use_id Email | Bearer Token | Validate succeeds | Validated Scuccessfully |
| Get User Information by Id | ‘/user/info’ | user_id | Bearer Token | Successfully retrieving user information | Successfully retrieved user information |
| Get User Information | ‘/user/find_by_email’ | Email | Bearer Token | Successfully retrieving | Successfully retrieved user |

| | | | | | |
|---|---|---|---|---|---|
| by Email | | | | user information | information |
| Get user access token | /user/check_token | Email | Bearer Token | Successfully get the access token | Successfully got the access token |

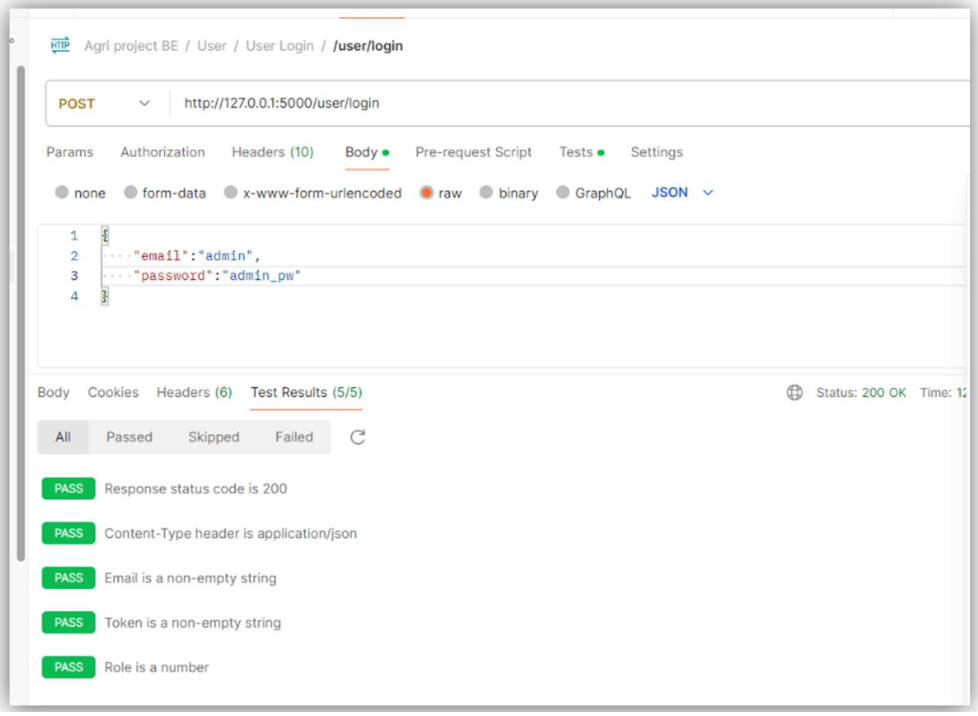



*Figure 5.2: Integration testing done using Postman for user login functionality - user login end point in API*

As this backend supports a lot of API endpoints, The following figures shows some of them are tested using Postman.

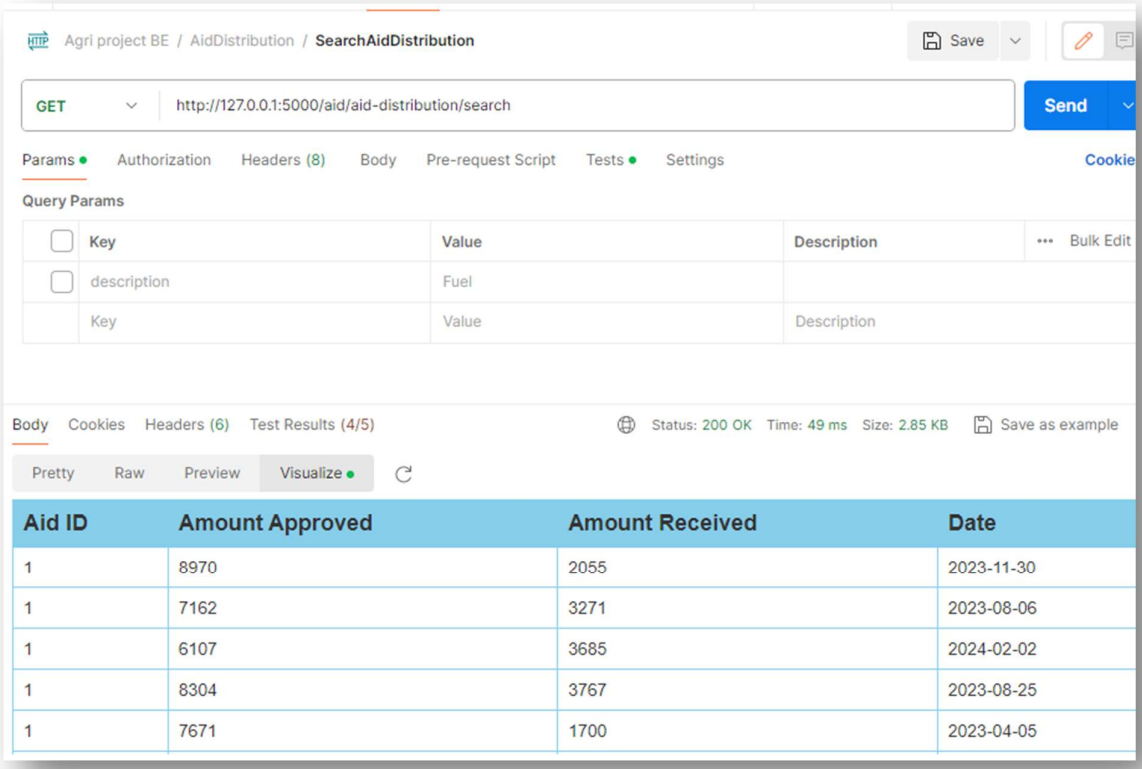



*Figure 5.3 Integration testing for Aid Distribution*

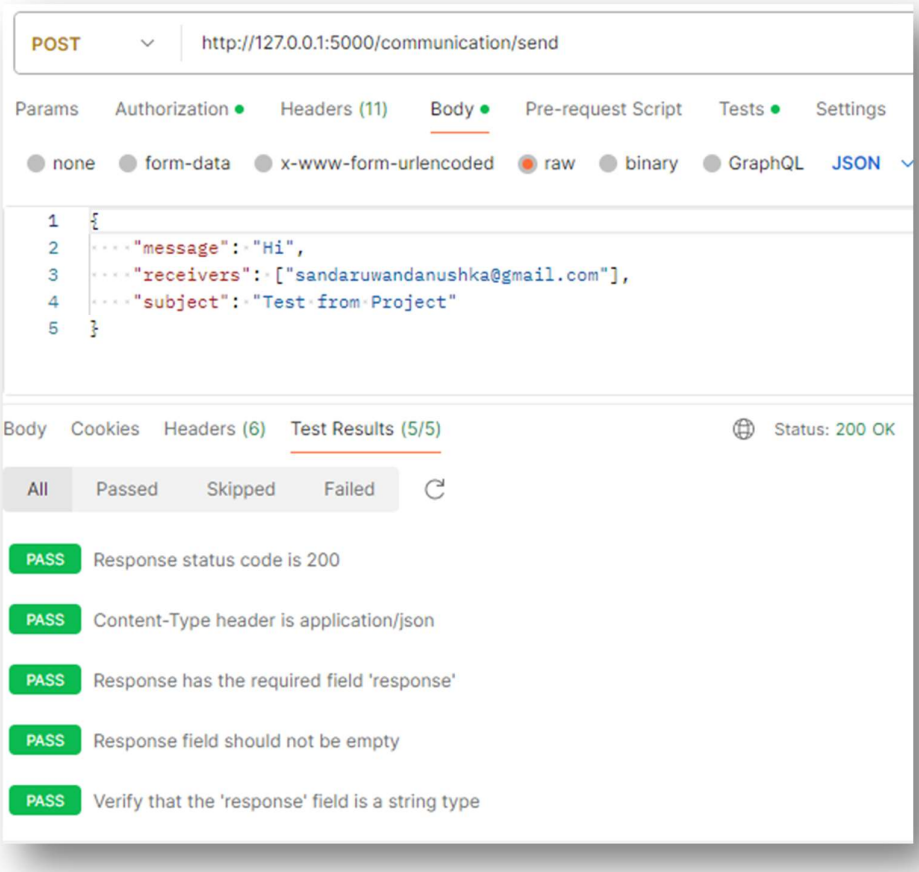



*Figure 5.4 Integration testing for sending emails*

All other major endpoints were meticulously tested to ensure seamless integration with the database using Postman. For test results, refer to the *Appendix I*.

## 5.5 Testing of Front-End (React Web Application)

### 5.5.1 Exploratory Testing – React web application

This involved navigating through the website just like a regular user would. By clicking on buttons, filling out forms, and interacting with different components, to ensure whether the all-features function smoothly as expects. Table 5.5 showcases the major test results with execution of this testing conducted manually.

*Table 5.5: Test Results of exploratory testing of web app*

| Id | Test case | Steps | Expected Results | Actual result |
|---|---|---|---|---|
| 1 | Access Web app as a generic user | Preview publicly available dash board,<br>Navigating through the web app,<br>See the public reports | Successfully interact with webapp as a generic user | Pass |
| 2 | Access Web app as a farmer | Preview publicly available dash board,<br>Navigating through the web app,<br>See the public reports,<br>Register in the system,<br>After validate by admin, publish advertisements | Successfully follow the steps and publish advertisements | Pass |
| 3 | Access Web app as a researcher | Preview publicly available dash board,<br>Navigating through the web app, | Successfully follow the steps and request data | Pass |

| | | | | |
|---|---|---|---|---|
| | | See the public reports,<br>Ask for non-public data through contact forum | | |
| 4 | Access Web app as an agriculture field officer | Preview publicly available dash board,<br>Do login,<br>Preview Field-officer dash board,<br>Manage agricultural data,<br>Generate reports,<br>Broadcast messages to registered farmers,<br>Approve advertisements | Successfully follow the steps | Pass |
| 5 | Access Web app as an admin | Preview publicly available dash board,<br>Do login,<br>Preview Admin dash board,<br>Insert Funding records,<br>Manage users,<br>Manage agricultural data,<br>Generate reports,<br>Broadcast messages,<br>Send requested data to researchers and maintain its records | Successfully follow the steps | Pass |

### 5.5.2 Cross-Browser Testing - React web application

To guarantee compatibility and consistent behavior across various platforms, the website was tested on different web browsers, including Chrome, Firefox, Edge as shown in the following figures. This comprehensive testing approach ensured that users could access and interact with the website seamlessly, regardless of their browser preference. See Figures 5.5 and 5.6.

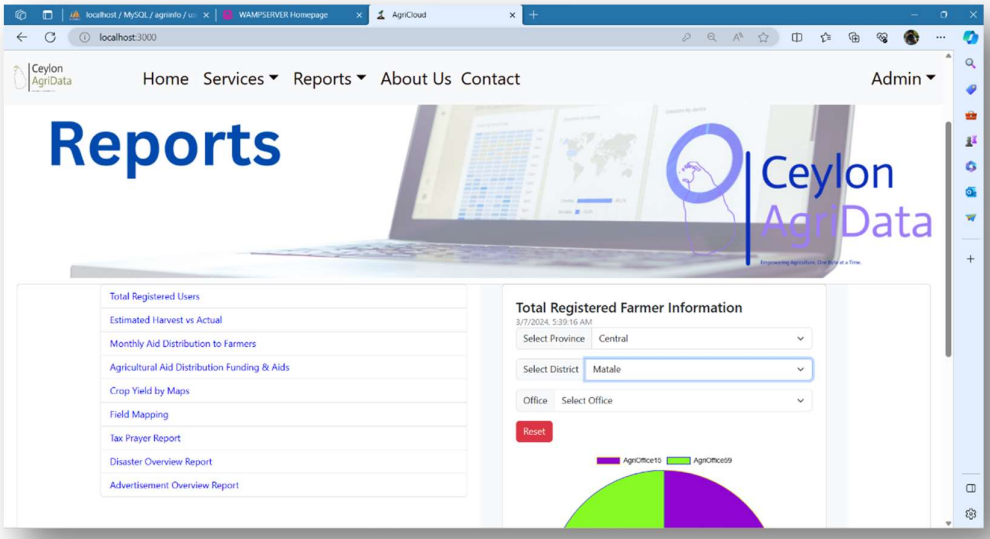



*Figure 5.5 Microsoft Edge testing*

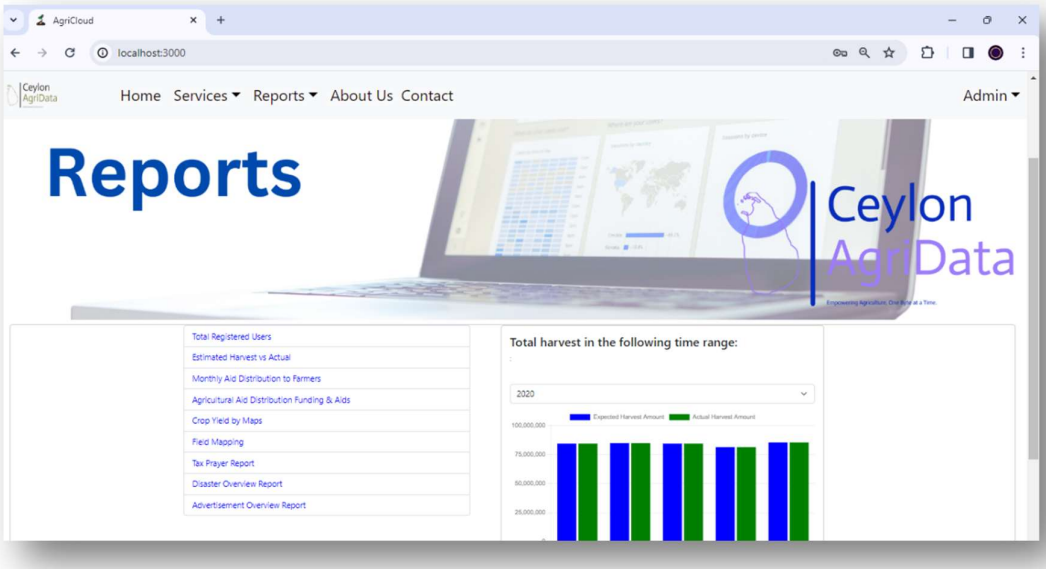



*Figure 5.6 Google Chrome testing*

### 5.5.3 End to end Testing - React web application

Complete End to end testing for the frontend along with locally hosted backend API was conducted for each feature implemented manually, ensuring the expected behavior of the frontend components of the react application with the backend API services. Finally, Backend was deployed and exposed publicly using Ngrok as shown in Figure 5.8 and deployed the frontend web application in another local machine after providing the correct configuration of the publicly exposed backend API service. Next, A comprehensive End to end testing was done starting from the User Authorization up to all features including report generation for each role type (farmer, administrator, field officer, researcher, generic user), free advertising support, email sending, data collection forms with geo location support and all data management operations including inserting updates, deletions and searches. In this testing we were able to identify the following issues shown in the Table 5.6 fixed them accordingly.

*Table 5.6: Part of identified major issues in end-to-end testing.*

| Issue | Identified Service | Description | Fix |
|---|---|---|---|
| When backend is not deployed, failing to handle the error properly. | Frontend React Web App | When user tries to logging, no responses from the system. | Added error handling & alert messages. |
| Some reports were shown regardless the role-based access | Frontend React Web App | When checking field officer reports, admin reports were shown | Fix the navigation based on the user role of the session of after a user logged in |
| Some reports data was not shown partially | Frontend React Web app - react charts | As we added pagination for data retrieval, only part of the data retrieve to the request. | Add mis-report API endpoint for report to process the data as in required format to the reports. |

| | | | |
|---|---|---|---|
| Some images are not rendered as expected | Frontend React Web app | Some images were too large or small when testing in different screens. | Added proper image sizes |
| Color combinations were not appropriate in the reports | Frontend React Web app - react charts | Some colors were not properly blend with the other depending on the place of rendering. | Tried to add color to make more user friendly |


*Figure 5.7 Locally deployed python flask backend interaction logs when testing*

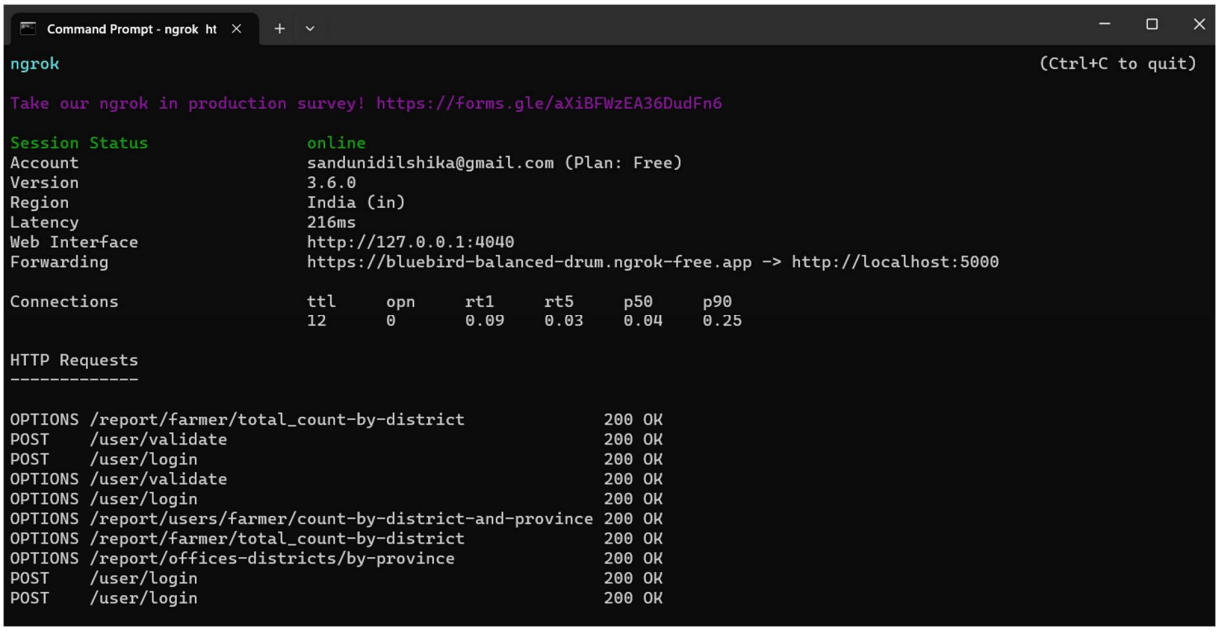



*Figure 5.8 : Ngrok logs when exposed API with the database publicly for testing*

## 5.6 User Evaluation

The system "Ceylon AgricData" (Mobile application and Webapp) underwent user acceptance testing (UAT) to gauge its reception among intended users. For UAT, commonly referred to as beta or end-user testing. A representative sample of users was chosen based on their roles. Administrators, agricultural field officers, farmers, and general users were targeted the testing that performed. These selected participants conducted an evaluation of the system, identifying both its strengths and weaknesses. Their assessments and feedbacks were captured through a specifically designed questionnaire form in Figure 5.9. A sample of eleven personal were participated in the evaluation and given their feedback through falling-out the questionnaire. The table below showcases the summarized outcomes of this user evaluation process. See *Appendix J* for the questionnaire.

With the limited resource and time allocated to this phase, the setup for the system built as in the end-to-end testing, as discussed above section, and provided to the users to interact with the system. Here, Mobile app was installed to android phones, a laptop with the locally deployed frontend application and another separate laptop with locally deployed backend API and exposed publicly using Ngrok static domain support to be used with the mobile apps and the frontend web app separately.

I had to meet physically the different roles of the system and provide the system for test. After presenting the system, provided the following survey form to the users. The different roles as per the selected samples were physically met and provided the system for testing. Thus, examined and interacted personas feedbacks were collected.

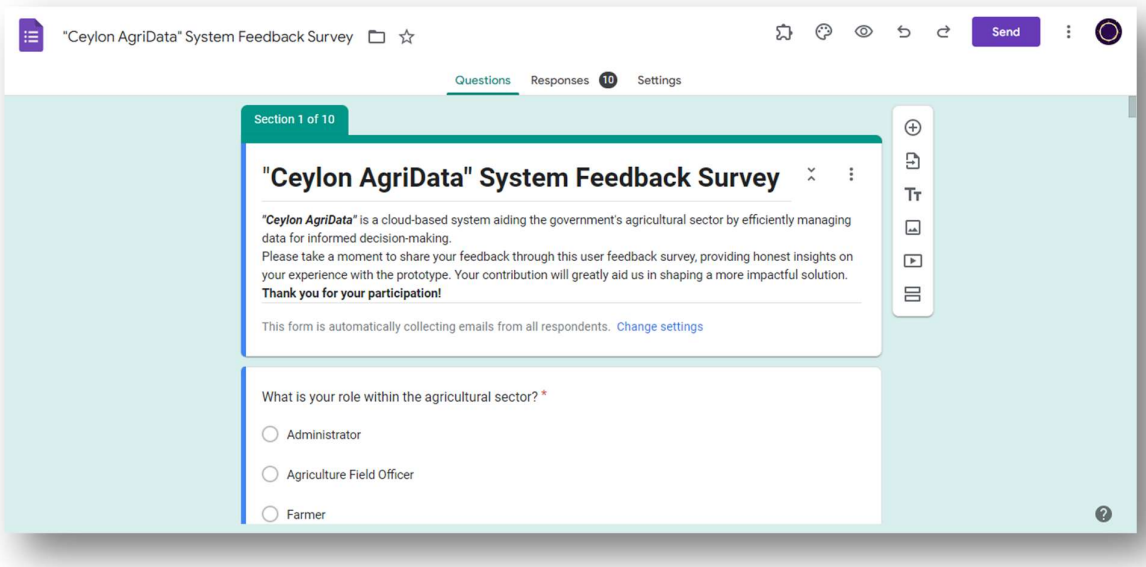



*Figure 5.9 System Feedback Survey*

### 5.6.1 Results of the Testing

The questionnaire was summarized to align with showcasing feedback from all roles together, facilitating a comprehensive evaluation. (see Table 5.6 )

- Total responded personas = 11

  Admin – 1

  Agriculture Field Officers -2

  Others are Farmers, Researchers, Generic Users

*Table 5.7 Summarized results*

| No | Statement | Strongly Agree | Agree | Neutral | Disagree | Strongly Disagree |
|---|---|---|---|---|---|---|
| 1 | **Registration Simplicity Rating** How would you assess the simplicity of the new user registration process on "Ceylon AgriData"? | 5 | 4 | 2 | 0 | 0 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | **Intuitiveness of Data Management**<br>How natural was the experience of inputting agricultural data on "Ceylon AgriData" | 0 | 2 | 1 | 0 | 0 |
| 3 | **Intuitiveness of Data Management**<br>How natural was the experience of updating agricultural data on "Ceylon AgriData" | 0 | 2 | 1 | 0 | 0 |
| 4 | **Intuitiveness of Data Management**<br>How natural was the experience of deleting agricultural data on "Ceylon AgriData" | 0 | 2 | 1 | 0 | 0 |
| 5 | **Intuitiveness of Data Management**<br>How natural was the experience of searching agricultural data on "Ceylon AgriData" | 2 | 1 | 0 | 0 | 0 |
| 6 | **Effectiveness of Reporting Tool**<br>How would you rate the efficacy of the reporting feature within "Ceylon AgriData" for generating detailed reports? | 4 | 3 | 3 | 0 | 0 |
| 7 | **Challenges in Report Configuration**<br>Did you face any difficulties while setting parameters for reports? Please describe your experience | 0 | 0 | 0 | 1 | 2 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 8 | **Effectiveness of Advertising Advertisements** How would you rate the efficacy of this feature within "Ceylon AgriData" | 1 | 6 | 4 | 0 | 0 |
| 9 | **General Satisfaction Level** How would you rate the UI design of "Ceylon AgriData" ? | 0 | 10 | 1 | 0 | 0 |
| 10 | **General Satisfaction Level** How would you describe your overall satisfaction with Ceylon AgriData's usability? | 0 | 10 | 1 | 0 | 0 |

There was a single administrator and two agricultural officers involved in the test, with the remainder being farmers and researchers. As such, the key functions can be assessed based on the contributions of these three officers, particularly in data management. The majority of researchers express satisfaction with the system, while farmers appreciate the free advertising service. Additionally, there was a suggestion to include language preferences, with Sinhala proposed as a secondary language. Overall, reports indicate positive outcomes.

## 5.7 Summary

Chapter 5 details the testing and evaluation of the "Ceylon AgriData" system. Various testing types were utilized, including unit testing, integration testing, end-to-end testing, and user acceptance testing. The testing methodology involved identifying test tasks, defining test cases, and conducting manual and automated testing. Results indicated success in building components and functions, though some issues were identified for improvement. User evaluation through acceptance testing gathered feedback from representative users. End-to-end testing of the React web application ensured expected behavior with backend services. Overall, the chapter showcases the system's reliability, functionality, and user satisfaction through rigorous testing and evaluation processes.

# Chapter 6 – Conclusion

This chapter provides an in-depth overview of the "Ceylon AgriData" cloud-based system, detailing its constraints, achievements, and drawbacks. It encompasses a comprehensive summary of the author's perspective and reflection on the system's development and implementation process.

## 6.1 Introduction

The agriculture sector of Sri Lanka holds significant importance in the country's economy. However, in recent times, various critical issues have emerged within the sector, leading to challenges and impediments. One of the primary identified issues is the lack of reliable availability of agriculture data. This absence of robust data-driven decision-making mechanisms has exacerbated the problems faced by the sector. Therefore, the establishment of a reliable and efficient data collection process aimed at supporting the agricultural sector becomes imperative. This system endeavors to address this crucial need by facilitating a streamlined data collection process.

## 6.2 Critical Assessment

The primary objective of the system is to facilitate informed decision-making within the agriculture sector through the utilization of collected data. Starting from the requirement analysis phase, all identified requirements were addressed to ensure comprehensive coverage. Subsequently, the project was initiated in key phases, including system design, implementation, and testing adopting iterative waterfall model.

The "Ceylon AgriData" system, operating as a cloud-based platform, is designed for the collection and presentation of agricultural data to support stakeholders involved in decision-making within the agriculture sector. Comprising a mobile application for field data collection by agriculture field officers and a web application developed with React for stakeholders such as agriculture officers, farmers, and researchers, the system aims to enhance efficiency and accessibility and transparency in the agricultural sector by replacing conventional paper-based methods of data acquiring and presentation.

Notably, the mobile application's major functionalities were integrated into the web application, allowing agriculture field officers to efficiently engage in data collection processes. They can utilize the mobile app during field visits for data collection, while the web application serves as a tool for office use. In cases where farmers visit agriculture offices to avail services, they can register with the system using the web application.

Although the system architecture was intentionally separated into distinct front-end and back-end components to facilitate ease of maintenance and scalability, the mobile application ensures operational efficiency for agriculture field officers. Additionally, registered farmers are provided with a feature for free advertising to promote their productions. Furthermore, governmental bodies can leverage the collected data for informed decision-making, particularly regarding price regulation.

To enhance communication among stakeholders and promote sectoral productivity, a message broadcasting service is employed for efficient dissemination of information. The implementation phase involved crafting of both front-end and back-end components, while testing procedures encompassed unit testing, integration testing, and user acceptance testing. The latter focused on user perspectives and assessed usability aspects. Through these measures, the system aims to fulfill its objective of supporting informed decision-making and enhancing productivity within the agriculture sector.

## 6.3 Lessons Learned

As a student pursuing a degree program, this project provided me with a valuable opportunity to apply theoretical knowledge in practical scenarios. Despite facing various challenges, I endeavored to complete the project within the stipulated time frame, adhering strictly to the Software Development Life Cycle (SDLC) methodology throughout the project duration.

Throughout the project, I gained invaluable insights into mobile application development, rapid development of React web applications, and the implementation of APIs. Additionally, I acquired proficiency in utilizing third-party libraries and conducting literature reviews in IT-related projects. Managing time effectively emerged as a crucial skill that I honed during the project, enabling me to navigate through the complexities of IT projects more efficiently.

Furthermore, I expanded my knowledge of different testing procedures applicable during the testing phases. Writing the thesis enabled me to develop the ability to succinctly summarize acquired knowledge and present it in an optimized flow, enhancing my communication and organizational skills in the process. Overall, this project served as a comprehensive learning experience, equipping me with valuable skills and insights that will undoubtedly prove beneficial in my future endeavors.

## 6.4 Problems Encountered During the Project

During the requirement gathering phase, I visited the agriculture office in Dodangoda, Kalutara district, and engaged in discussions with agriculture officers. However, as time passed, they informed that it takes much time to provide the necessary data for the system. Therefore, had to continue the project with a planned time frame with limited information gathered from the agriculture office. And also, I had to use mock data in the system.

I had to do a limited user acceptance testing with just few identified roles without any intervention to official connections.

Additionally, the project utilized updated technologies, which required a substantial amount of time to learn before implementation could begin. This learning curve proved to be a major hurdle, especially considering the limited time available for project completion. As someone not coming from an IT background, acquiring proficiency in these new technologies took longer than anticipated, further exacerbating the time constraints.

Despite planning to incorporate an SMS gateway into the project, the need to purchase such gateways, coupled with the constraints of time and resources, presented significant challenges. These obstacles were particularly daunting given the individual nature of the project.

## 6.5 Potential Future Work

1. **Improve Security**

Enhancing security measures is essential for safeguarding sensitive data. While the system currently employs encrypted usernames and passwords, as well as JWT access tokens for user

validation and session maintenance, implementing OTP functionality could provide an additional layer of security. Furthermore, integrating a separately available user management system with an identity service can further fortify the security infrastructure of the system.

2. **Enhance Frontend Usability**

Improving frontend usability is crucial for enhancing user satisfaction and facilitating smoother interactions. Utilizing advanced CSS techniques and incorporating user feedback can greatly enhance the user experience, making the system more intuitive and user-friendly.

3. **Expand Report Functionalities**

Expanding the range of report types can provide users with more comprehensive insights and analysis capabilities. By incorporating additional reporting functionalities, users can access a wider range of data representations, enabling more informed decision-making within the agriculture sector.

4. **Implement News Updates**

Keeping users informed with relevant news updates directly within the web application can enhance user engagement and provide valuable insights into industry trends and developments. Integrating a news update feature can help users stay up-to-date with the latest information, enriching their overall experience with the system.

5. **Agriculture Knowledge Sharing Portal**

Integrating a knowledge-sharing portal within the system can facilitate the exchange of valuable information and insights among stakeholders in the agriculture sector. By providing a platform for knowledge dissemination, users can access and share expertise, fostering collaboration and innovation within the agricultural community.

6. **Robust Marketplace Integration**

Incorporating a robust marketplace integration feature enables users to directly engage in buying and selling agricultural products within the system. By seamlessly integrating marketplace functionalities, users can leverage the platform for efficient and convenient transactions, enhancing productivity and profitability within the agriculture sector.

# References


Aaron O'Nell. 2023. *Share of economic sectors in the GDP in Sri Lanka 2021* [Online]. Available: https://www.statista.com/statistics/728539/share-of-economic-sectors-in-the-gdp-in-sri-lanka/ [Accessed 18-06-2023].

Agrible-Morning Farm Report. *Agrible – Morning Farm Report* [Online]. Available: https://u.osu.edu/agsoftwarelibrary/2018/03/21/agrible-morning-farm-report/ [Accessed 2023].

Agrimanager. *Agrimanager* [Online]. Available: https://www.getapp.com/industries-software/a/agrimanager/ [Accessed 18-06-2023].

Agrithing. *AgriThing* [Online]. Available: https://agrithing.com/ [Accessed 19-06-2023].

Agriwebb. *AgriWebb* [Online]. Available: https://www.agriwebb.com/ [Accessed 2023].

Agsense. 2003. *AgSense* [Online]. Available: https://www.agsense.com/ [Accessed 2003].

Awhere. *aWhere* [Online]. Available: https://www.climateshot.earth/awhere [Accessed 2023].

Bhatnagar, V. 2015. A comprative study of sdlc model. *IJAIEM,* 4**,** 23-29.

Bushelfarm. *BUSHEL* [Online]. Available: https://bushelpowered.com/ [Accessed 2023].

Cheema, M. J. M. & Khan, M. A. 2019. Information Technology for Sustainable Agriculture. *In:* Farooq, M. & Pisante, M. (eds.) *Innovations in Sustainable Agriculture.* Cham: Springer International Publishing.

Christine Zhenwei Qiang, Siou Chew Kuek, Andrew Dymond & Steve Esselaar 2012. Mobile Applications for Agriculture and Rural Development. *In:* ICT SECTOR UNIT, W. B. (ed.). World Bank

Climate Corporation. *Climate FieldView* [Online]. [Accessed 2023].

Coconut Cultivation Board. 2023. *Coconut App* [Online]. Available: https://play.google.com/store/apps/details?id=zincat.net.cocoguru&hl=en&gl=US [Accessed 18-06-2023].

Conservis. 2008. *Conservis* [Online]. Available: https://conservis.ag/ [Accessed 2023].

Cropin. 2010. *Cropin* [Online]. Available: https://www.cropin.com/ [Accessed 19-06-2023].
Dennis, A. 2012. *Systems Analysis and Design*, Wiley Publishing.

Department of Agriculture. 2021. *Krushi Advisor* [Online]. Available: https://play.google.com/store/apps/details?id=com.prasadbandra.krushiadvisor&hl=en&gl=US [Accessed].

FAO, F. A. A. O. O. T. U. N. 2023. The State of Food and Agriculture 2023. Rome.

Farm Force. *Farm Force* [Online]. Available: https://farmforce.com/ [Accessed 19-06-2023].
Govi Mithuru. 2015. *Govi Mithuru* [Online]. Available: https://www.dialog.lk/govi-mithuru/ [Accessed 19-06-2023].

Granular, I. *Granular* [Online]. Canada, US. Available: https://www.farms.com/agriculture-apps/technology/granular [Accessed 17-06-2023 2023].

Helawiru. *Helaviru Platform* [Online]. Available: https://www.helaviru.lk/ [Accessed 19-06-2023].

Hillar, G. C. 2015. *Learning Object-Oriented Programming*, Packt Publishing.
Jayathilake, H., Jayaweera, B. & Waidyasekera E. 2010. ICT Adoption and Its' Implications for Agriculture in Sri Lanka. *Journal of Food and Agriculture,* 1(2)**,** 54-63.

KOMA LABS. 2020. *Agro Life Sri Lanka* [Online]. Available: https://play.google.com/store/apps/details?id=io.ionic.prog5e986cedd814f9698bb3adac&hl=en&gl=US [Accessed 16-06-2023].

Manoj Thibbotuwawa. 2021. *Leveraging technological innovations can help overcome growth constraints and increase agriculture's economic contribution* [Online]. Available: https://development.asia/insight/why-transition-smart-farming-critical-sri-lanka [Accessed 20-06-2023].

Ministry of Agriculture , M. 2023. *Overview* [Online]. Available: agrimin.gov.lk/web/index.php/en/about-us/overview123 [Accessed 2023].

MÜNCH, T. 2022. System Architecture Design. *In:* MÜNCH, T. (ed.) *System Architecture Design and Platform Development Strategies: An Introduction to Electronic Systems Development in the Age of AI, Agile Development, and Organizational Change.* Cham: Springer International Publishing.

My Agri. 2021. *My Agri* [Online]. Available: https://play.google.com/store/apps/details?id=com.Erlanggastudio.MyAgri&hl=en&gl=US [Accessed 2023].

Ojha, T., Misra, S. & Raghuwanshi, N. 2015. Wireless Sensor Networks for Agriculture: The State-of-the-Art in Practice and Future Challenges. *Computers and Electronics in Agriculture,* 118.

Petrillo, F., Merle, P., Moha, N. & Gueheneuc, Y.-G. 2016. *Are REST APIs for Cloud Computing Well-Designed? An Exploratory Study*.

Rathod, M., Shivaputra, A., Umadevi, H., Kenchappa, N. & Selvakumar Periyasamy, D. 2022. Cloud Computing and Networking for SmartFarm AgriTech. *Journal of Nanomaterials,* 2022**,** 1-7.

Richards, M. 2022. *Software Architecture Patterns*, O'Reilly Media, Inc.
Singh, S., Chana, I. & Buya, R. 2020. Agri-Info: Cloud Based Autonomic System for Delivering Agriculture as a Service. *Internet of Things,* 9**,** 100131.

Volodymyr Ahafonkin. 2010. *Leaflet - a JavaScript library for interactive maps* [Online]. Available: https://leafletjs.com/ [Accessed].

Worldbank. 2023. *Agriculture and Food* [Online]. Available: https://www.worldbank.org/en/topic/agriculture/overview#1 [Accessed 2023].

# Appendixes

- Appendix A
- Appendix B
- Appendix C
- Appendix D
- Appendix E
- Appendix F
- Appendix G
- Appendix H
- Appendix I
- Appendix J
- Appendix K
- Appendix L

# Appendix A

# MIT3201 – Individual Project in MIT Degree Program

## MIS Report

**Name** **:** L.R.S.D.Rathnayake

**IndexNo** 20550839

**Supervisor Signature:**

**Supervisor Name:** Prof. M.G.N.A.S. Feranando

# Management Information System (MIS) Report Templates for ‘AgriCloud’ System

### Executive Summary

The ‘AgriCloud’ System introduces a comprehensive suite of Management Information System (MIS) report templates designed to facilitate data-driven decision-making in the agricultural sector. These templates encompass a broad spectrum of functionalities, including user registration, aid distribution, disaster analysis, tax compliance, communication effectiveness, and marketplace engagement. Through detailed data collection, analysis, and representation, these reports aim to enhance the efficiency of agricultural management and operations, ensuring accountability, transparency, and informed strategic planning.

## MIS Report Template 1 : Registered User Report

This report provides detailed information about registered users in the ‘AgriCloud’ system, including agriculture officers and farmers. It includes user IDs, names, contact details, registration dates, and assigned regions for agriculture officers. It aims to facilitate efficient management of user data.

**Data Summary:**

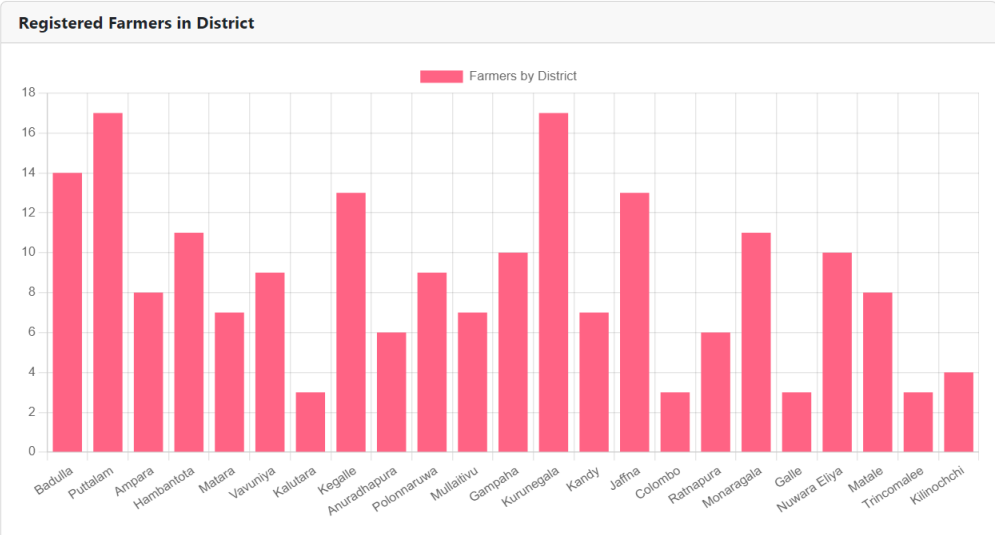
- Total number of registered users
- Breakdown of registered users by role (agricultural officers, farmers)
- Profile information of each user (name, contact details, role, assigned region/area)
- Date of registration

**Analysis:**

- Comparison of user distribution across different regions or districts
- Identification of any discrepancies or irregularities in user data

**Representation:**

- Detailed tables containing user information
- Visualizations such as pie charts or bar graphs depicting user distribution by region or role



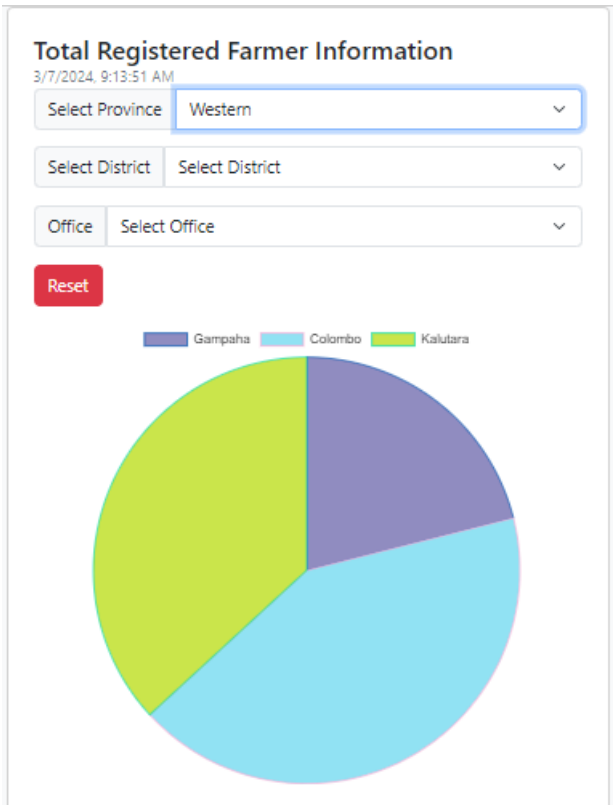



*Figure1:Total Registered Users in the system*

| User ID | First Name | Last Name | NIC | Role | Actions |
|---|---|---|---|---|---|
| 1 | Farmer1 | Last1 | 199000000001 | 5 | |
| 2 | Farmer2 | Last2 | 199000000002 | 5 | |
| 3 | Farmer3 | Last3 | 199000000003 | 5 | |
| 4 | Farmer4 | Last4 | 199000000004 | 5 | |
| 5 | Farmer5 | Last5 | 199000000005 | 5 | |
| 6 | Farmer6 | Last6 | 199000000006 | 5 | |
| 7 | Farmer7 | Last7 | 199000000007 | 5 | |
| 8 | Farmer8 | Last8 | 199000000008 | 5 | |
| 9 | Farmer9 | Last9 | 199000000009 | 5 | |
| 10 | Farmer10 | Last10 | 199000000010 | 5 | |
| 11 | Farmer11 | Last11 | 199000000011 | 5 | |

*Figure 2: Total Registered user details*

# MIS Report Template 2 : Aid Distribution Report

This report offers a detailed look at the distribution of various forms of aid provided to farmers. It covers several types of assistance, such as fuel, fertilizer, pesticides, and financial support. Key information included in the report comprises the date of distribution, the specific type of aid given, the amount distributed, details of the recipients (including farmer ID), and the region where the aid was distributed. The report assists in monitoring aid distribution activities and evaluating the effectiveness of support programs. It ensures transparency and accountability in aid allocation and distribution processes.

**Data Summary:**

- The overall quantity of aid provided, including fuel, fertilizer, pesticides, and financial support.
- A detailed breakdown showing the type of aid and the quantities distributed.
- Distribution timeline and frequency
- Details about the recipients, including farmers' names, their locations, and the specific aid they received.

**Analysis**:

- Assessment of aid distribution patterns and trends over time
- Evaluation of aid utilization effectiveness and impact on agricultural productivity
- Identification of areas or demographics with higher aid requirements

**Representation**:

- Comprehensive tables that lay out all the data related to the distribution of aid.
- Visual graphs to help show the trends and patterns in aid distribution, making it easier to understand at a glance.

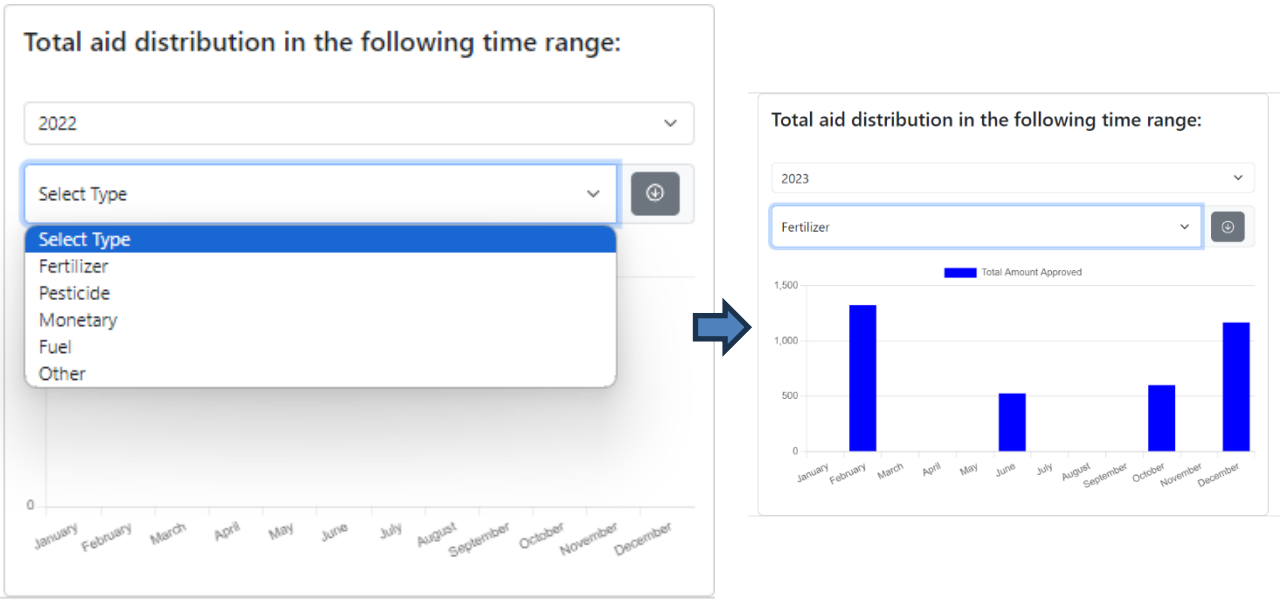
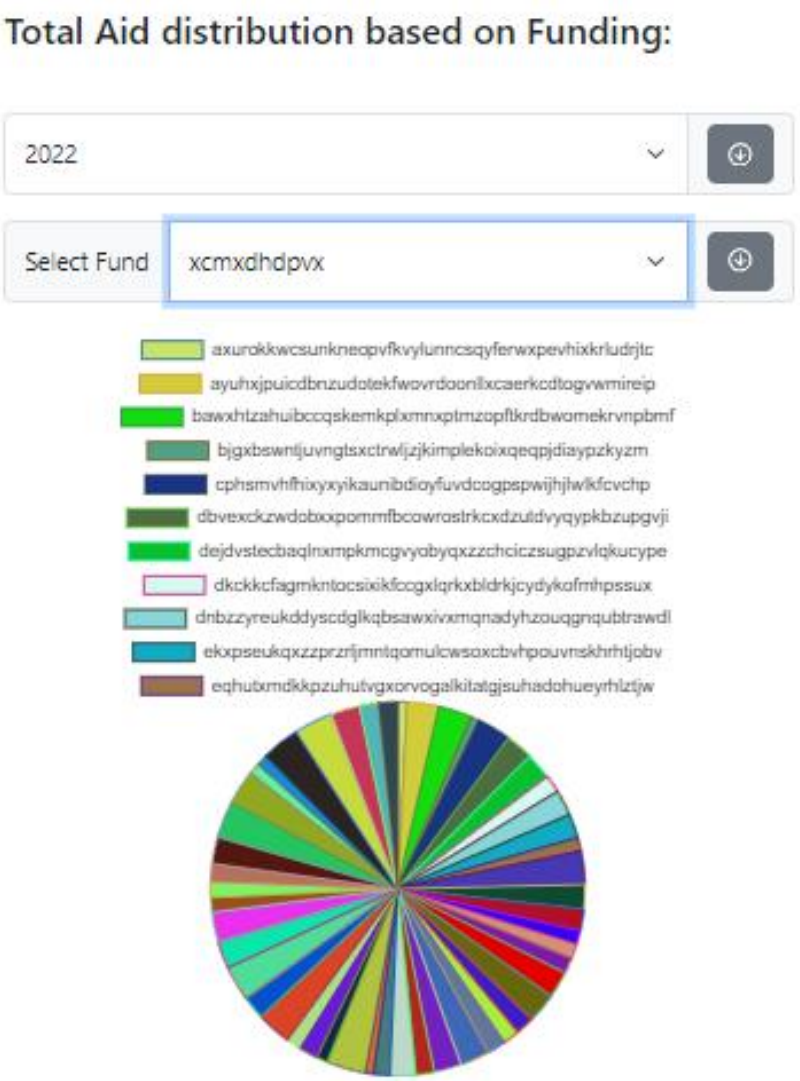



*Figure 3: Aid Distribution*

*Figure 4: Aid Funding Report*



# MIS Report Template 3 : Crop Yield Analysis Report

This report offers an in-depth analysis of crop yield data, organized by crop type, geographical region, and growing season. It encompasses vital details such as the specific type of crops, the quantity of yield, the regions where these crops were grown, and the time of year they were harvested. The primary goal is to uncover trends, patterns, and the key factors that affect crop productivity, thereby aiding in making well-informed agricultural decisions.

**Data Summary:**

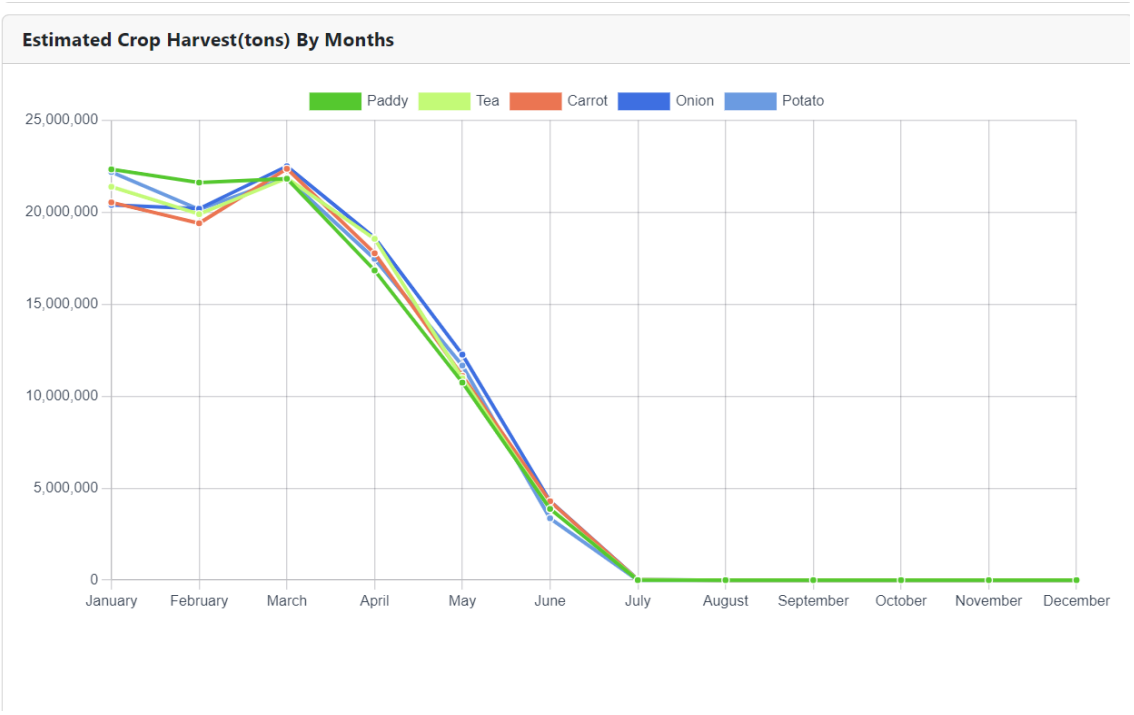
- The total amount of crop yield data gathered.
- A detailed division of crop yields according to the type of crop, the region it was grown in, and the season it was harvested.
- The average yield per hectare for each type of crop.
- A comparative study examining how yields vary across different seasons and regions.

**Analysis:**

- Identification of high-performing and underperforming crops and regions
- Evaluation of seasonal variations and their impact on crop productivity

**Representation:**

- Comprehensive tables that organize crop yield data by type, region, and season, providing a clear overview.
- Various charts and graphs that visually depict the trends and fluctuations in crop yields, making it easier to spot patterns and anomalies.



| Farm ID | Crop ID | Name | Location | Estimated Harvest | Agri Year | Quarter | Estimated Harvesting Date | Harvested Date | Harvested Amount | Recorded Date | Actions |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | df | View Map | 1234 | 2323 | 21 | 2024-02-19 | | | 2024-02-16 | |
| 1 | 1 | df | View Map | 1234 | 2323 | 21 | 2024-02-19 | | | 2024-02-16 | |
| 1 | 1 | rice | View Map | 1234 | 2024 | 2 | 2024-02-20 | | | 2024-02-16 | |

1

*Figure 6: Harvesting details*

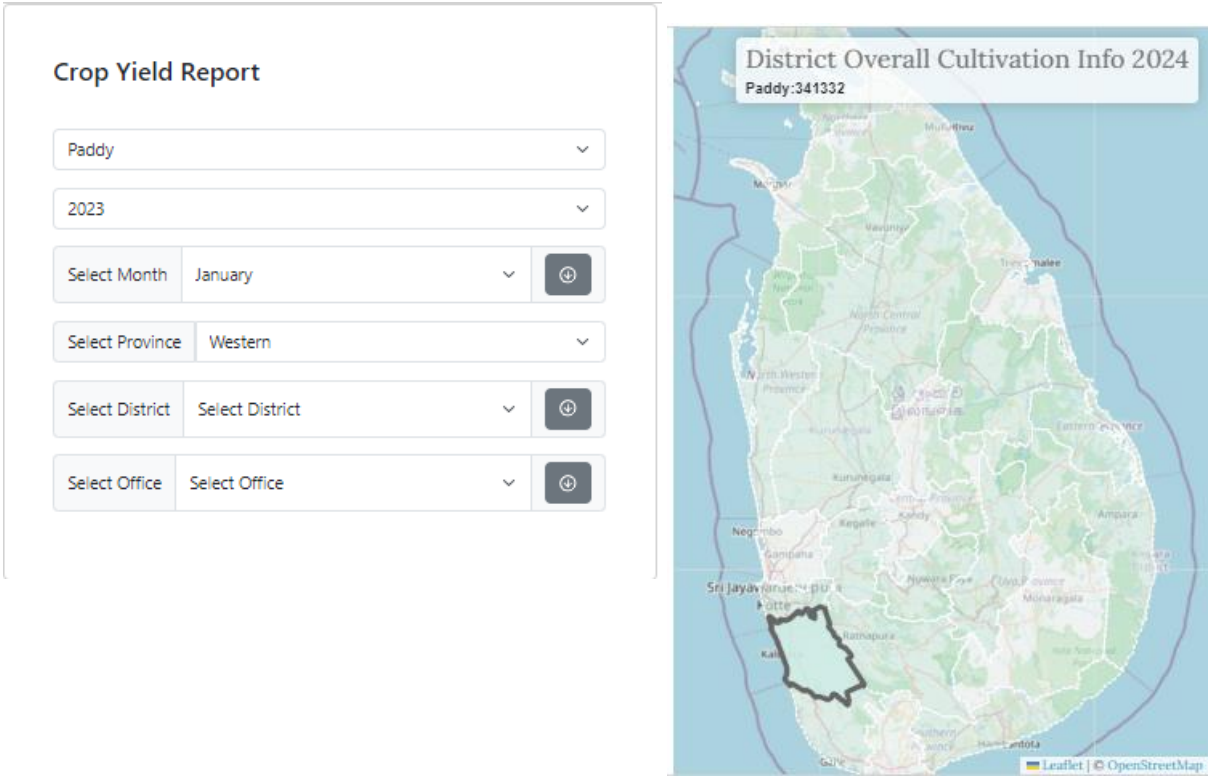
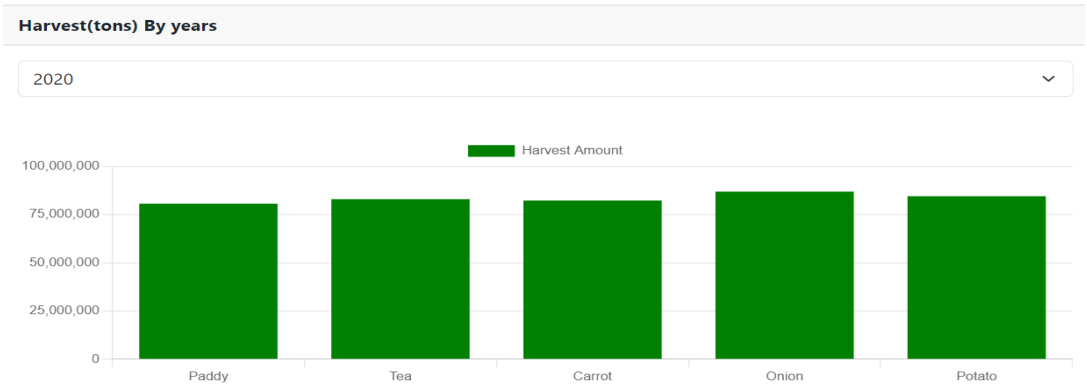



*Figure 7: Crop Yield report*



## MIS Report Template 4 : Field Mapping Overview Report

This report provides an overview of field mapping activities conducted within the 'AgriCloud' system. It includes details such as field ID, location coordinates, assigned officer, and mapping date. The report facilitates monitoring of field mapping progress and ensures accurate spatial data management. It helps to Assessment of the impact of field mapping on resource allocation and decision-making processes

**Data Summary:**

- The total count of fields that have been mapped.
- How these mapped fields are distributed across different regions and according to crop types.

**Analysis**:

- Evaluation of field mapping coverage across different agricultural regions
- Identification of areas with incomplete or outdated mapping data

**Representation:**

- Maps and other visual aids that show the extent of field mapping coverage and how mapped fields are distributed, helping to visualize the scope and scale of mapping activities.

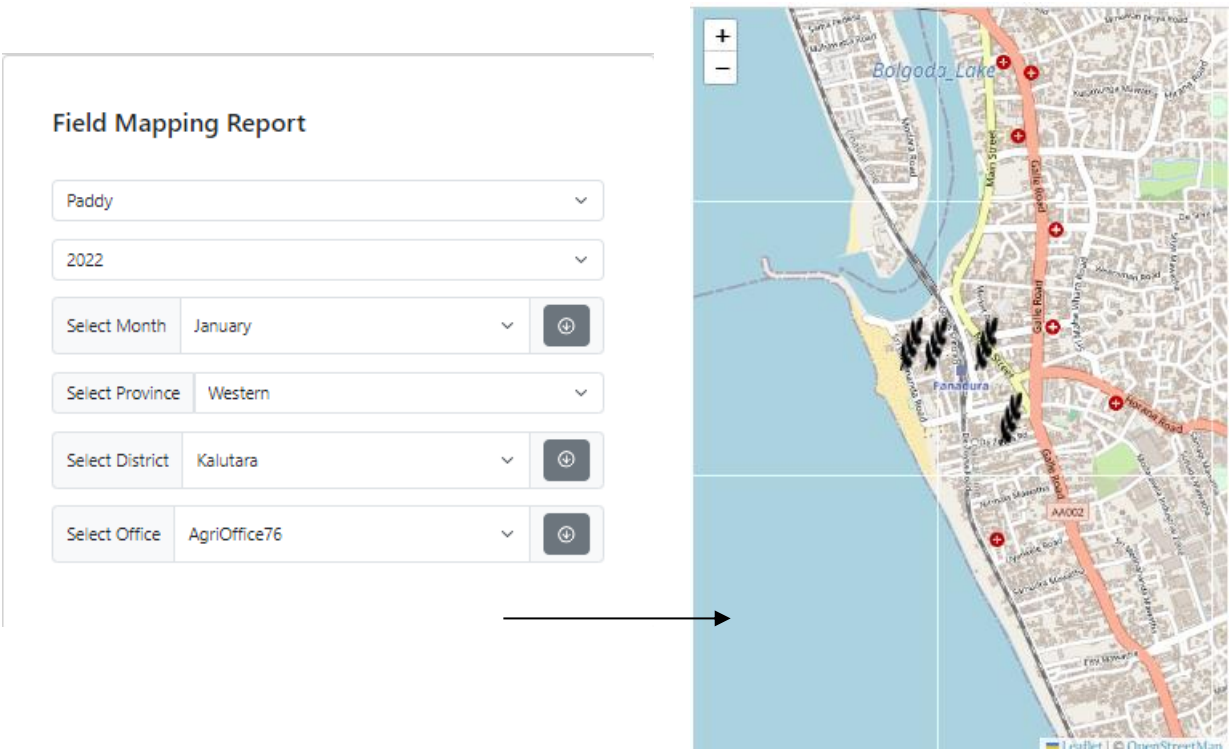



*Figure 8: Field Mapping*

## MIS Report Template 5: Acre Tax Tracking Report

This report tracks acre tax payments recorded by farmers in the AgriCloud system. It includes key details such as the farmer's unique identifier (farmer ID), the amount paid, the date of payment, and the geographical region. The main objectives of this report are to oversee tax compliance, enhance revenue collection, and improve financial management practices. Additionally, it aims to promote transparency and accountability in the tax collection process.

**Data Summary:**

- A detailed breakdown of tax payees, categorized by farmers in regions

**Analysis**:

- Comparison of tax payees (farmers) distribution across different regions or districts

**Representation:**

- Comprehensive tables that detail acre tax payment information, facilitating easy review and analysis.
- Visual representations, such as charts and graphs, that depict trends in tax payments and highlight variations in compliance rates across different regions or times.

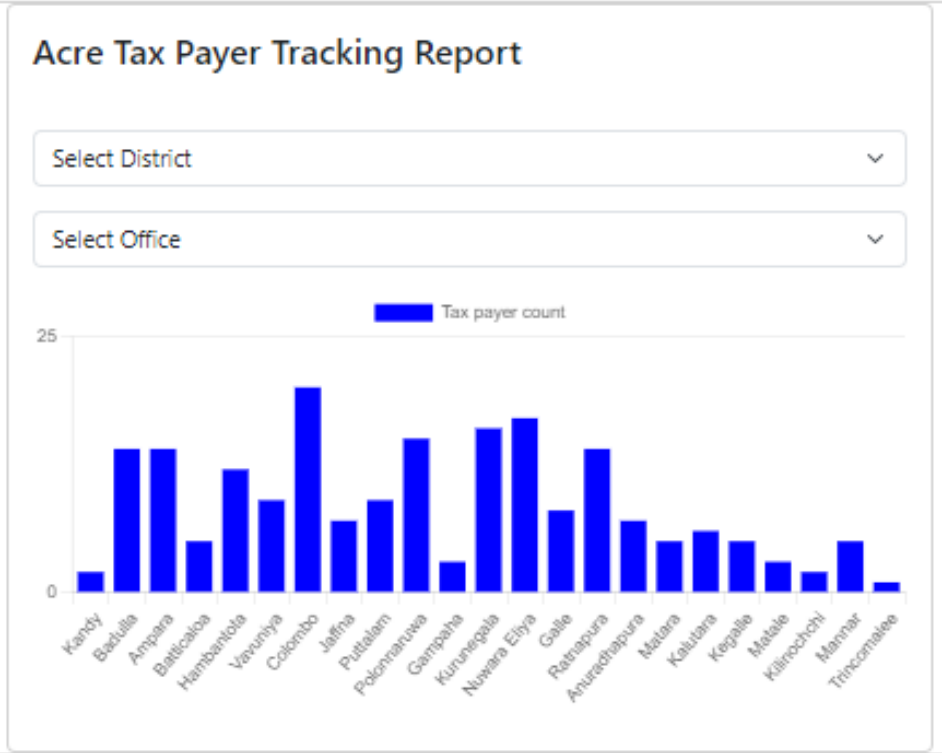



*Figure 9: Acre tax payer tracking report*

## MIS Report Template 6: Broadcast Message Report

This report compiles and analyzes the broadcast messages sent within the AgriCloud system. It captures essential information about each message, including its content, the sender's details, the type of recipients targeted, and the time it was sent. The focus is on maintaining a comprehensive record of messages broadcasted by agriculture officers to various stakeholders.

**Data Summary:**

The aggregate count of broadcast messages that have been sent.

A categorized summary of these messages, detailing the sender, the recipient group, and the type of content shared.

**Analysis:**

- Evaluation of message content and relevance to target audience

**Presentation**:

- Comprehensive logs that detail the activities related to broadcast messages, including sender, content, and recipient information, presented in detailed tables for clarity and ease of analysis.

| Mail ID | Sender | Message | Status | Sent To | Response |
|---|---|---|---|---|---|
| 1 | sandunidilshika@gmail.com | test message | | email12@example.com | {"id": "18e0e0d38efc8182", "threadId": "18e0e0d38efc8182", "labelIds": ["SENT"]} |
| 2 | sandunidilshika@gmail.com | test message | | email24@example.com | {"id": "18e0e0d4069f589a", "threadId": "18e0e0d4069f589a", "labelIds": ["SENT"]} |
| 3 | sandunidilshika@gmail.com | test message | | email35@example.com | {"id": "18e0e0d4768dda16", "threadId": "18e0e0d4768dda16", "labelIds": ["SENT"]} |
| 4 | sandunidilshika@gmail.com | test message | | email52@example.com | {"id": "18e0e0d4cafe8183", "threadId": "18e0e0d4cafe8183", "labelIds": ["SENT"]} |
| 5 | sandunidilshika@gmail.com | test message | | email56@example.com | {"id": "18e0e0d540b050dc", "threadId": "18e0e0d540b050dc", "labelIds": ["SENT"]} |
| 6 | sandunidilshika@gmail.com | test message | | email65@example.com | {"id": "18e0e0d5ccdbc2d9", "threadId": "18e0e0d5ccdbc2d9", "labelIds": ["SENT"]} |
| 7 | sandunidilshika@gmail.com | test message | | email67@example.com | {"id": "18e0e0d635df3015", "threadId": "18e0e0d635df3015", "labelIds": ["SENT"]} |
| 8 | sandunidilshika@gmail.com | test message | | email83@example.com | {"id": "18e0e0d69801f094", "threadId": "18e0e0d69801f094", "labelIds": ["SENT"]} |
| 9 | sandunidilshika@gmail.com | test message | | email84@example.com | {"id": "18e0e0d723ccd477", "threadId": "18e0e0d723ccd477", "labelIds": ["SENT"]} |

*Figure 10:Message broadcasting logs*

# MIS Report Template 7: Disaster Analysis Report

This report delves into agricultural disaster data within the AgriCloud system, with the objective of evaluating the impact of such disasters on crop yields and overall agricultural productivity. It aims to provide valuable insights that can inform disaster preparedness and response strategies effectively.

**Data Summary:**

- A catalog of the types of agricultural disasters encountered and their occurrence frequency.
- Detailed assessments of the damage and losses incurred from each type of disaster.
- The geographical spread of areas affected by disasters.

**Analysis:**

- An examination of the agricultural systems' vulnerabilities and their resilience against various disaster scenarios.
- Identification of high-risk areas and crops susceptible to disaster damage.

**Representation:**

- Detailed reports on disaster events, including damage assessments and response activities
- Visual aids such as maps and charts to depict areas hit by disasters and the extent of crop damage, facilitating a clearer understanding of the impact and scope.

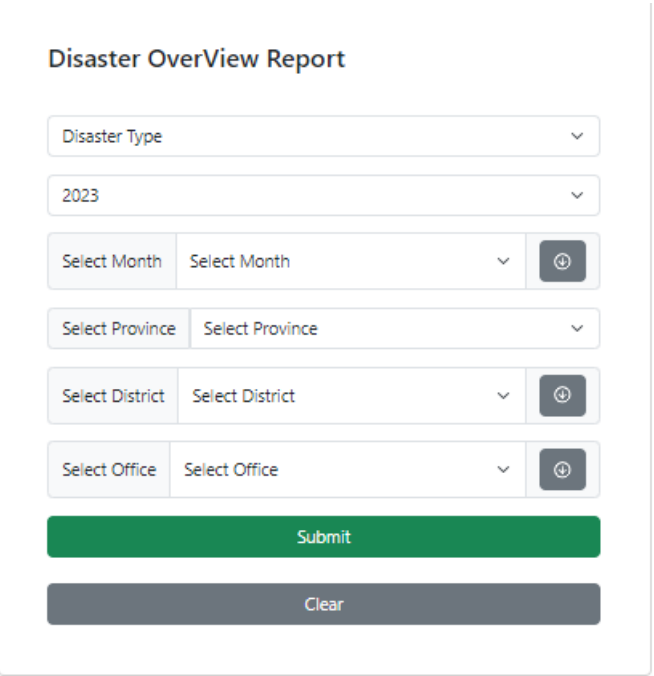



*Figure 0.1: Disaster overview generating filter form*

## MIS Report Template 8: Free Advertising Service Engagement Report

his report examines how registered users, including farmers and vendors, interact with the marketplace feature on the AgriCloud system. It seeks to understand the marketplace's role in bridging the gap between farmers and potential buyers, evaluating its efficiency and effectiveness.

**Data Summary:**

- The count of farmers and vendors actively participating in the marketplace.
- The variety and distribution of products listed, including details on advertisements posted.

**Analysis:**

- An analysis of how the marketplace is adopted and utilized by its users, identifying patterns and trends in usage.
- An evaluation of the marketplace's success in ensuring fair pricing and efficient transactions between sellers and buyers.

**Representation:**

- A detailed account of the advertisements posted, including information on the types of products, their pricing, and the total volume of listings. This helps to provide a comprehensive overview of the marketplace's activity and offerings.

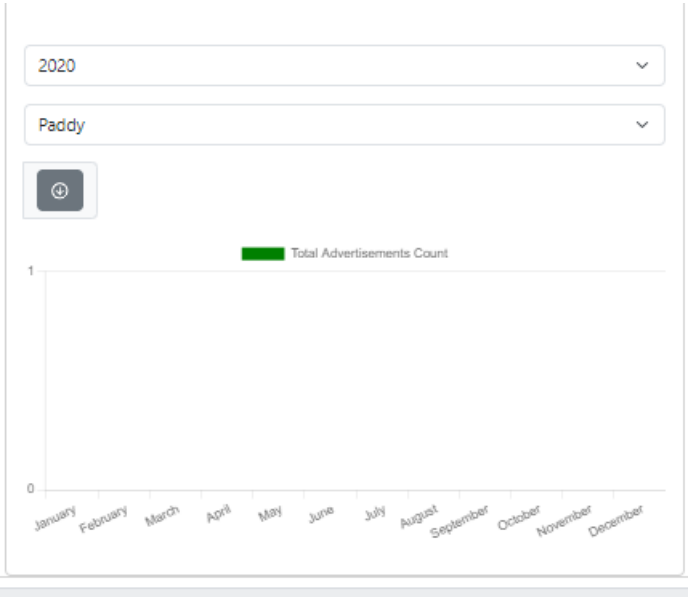



*Figure 0.1: Advertisement overview report*

## Conclusion

In conclusion, these Management Information System (MIS) Report Templates serve as essential tools for the ‘AgriCloud’ system, offering structured and detailed insights into various operational aspects. From tracking registered user engagement and analyzing crop yields to monitoring disaster impacts and assessing marketplace dynamics, each template is designed to facilitate informed decision-making and strategic planning. By systematically collecting, analyzing, and representing data, these reports enable stakeholders to identify trends, evaluate effectiveness, and identifying areas for improvement. Ultimately, the effective use of these templates will enhance operational efficiencies, support sustainable agricultural practices, and foster a more resilient and productive agricultural sector.

** Evidences were taken with several mock data inserted in database.

# Appendix B

Database Models int the project are as follows.

```
# Role table
class Role(db.Model):
    __tablename__ = 'role'
    role_id = Column(Integer, primary_key=True)
    role_name = Column(String(100))
    role_description = Column(String(100))
# Contact table
class Contact(db.Model):
    __tablename__ = 'contact'
    contact_id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey('user.user_id'))
    number = Column(String(100))
    area_code = Column(String(100))
    user = relationship("User", backref="contacts")
# Address table
class Address(db.Model):
    __tablename__ = 'address'
    address_id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey('user.user_id'))
    city = Column(String(100))
    town = Column(String(100))
    street = Column(String(100))
    home_no = Column(String(100))
    home_name = Column(String(100))
    user = relationship("User", backref="addresses")
# AgricultureOfficer table
class AgricultureOfficer(db.Model):
    __tablename__ = 'agriculture_officer'
```

```
    user_id = Column(Integer, ForeignKey('user.user_id'), primary_key=True)
    employee_id = Column(Integer)
    managed_by_employee_id = Column(Integer)
    agri_office_id = Column(Integer)
    service_start_date = Column(Date)
    field_area_id = Column(Integer)
    user = relationship("User", backref="agriculture_officers")
# AgriOffice table
class AgriOffice(db.Model):
    __tablename__ = 'agri_office'
    agri_office_id = Column(Integer, primary_key=True)
    name = Column(String(100))
    city = Column(String(100))
    province = Column(String(100))
    district = Column(String(100))
# FieldArea table
class FieldArea(db.Model):
    __tablename__ = 'field_area'
    field_area_id = Column(Integer, primary_key=True)
    agri_office_id = Column(Integer,
ForeignKey('agri_office.agri_office_id'))
    name = Column(String(100))
    agri_office = relationship("AgriOffice", backref="field_areas")
# Reports table
class Reports(db.Model):
    __tablename__ = 'reports'
    report_id = Column(Integer, primary_key=True)
    category = Column(String(100))
    date = Column(Date)
    time = Column(String(100))
    user_id = Column(Integer, ForeignKey('user.user_id'))
    user = relationship("User", backref="reports")
```

```
# Farmer table
class Farmer(db.Model):
    __tablename__ = 'farmer'
    user_id = Column(Integer, ForeignKey('user.user_id'),primary_key=True)
    assigned_office_id = Column(Integer,
ForeignKey('agri_office.agri_office_id'))
    assigned_field_area_id = Column(Integer,
ForeignKey('field_area.field_area_id'))
    updated_by = Column(Integer)
    added_by = Column(Integer)
    registered_date = Column(Date)
    tax_file_no = Column(String(100))
    user = relationship("User", backref="farmers")
    assigned_office = relationship("AgriOffice",
foreign_keys=[assigned_office_id])
    assigned_field_area = relationship("FieldArea",
foreign_keys=[assigned_field_area_id])
# Login table
class Login(db.Model):
    __tablename__ = 'login'
    user_id = Column(Integer, ForeignKey('user.user_id'),primary_key=True)
    username = Column(String(100))
    encoded_pw = Column(String(100))
    user = relationship("User", backref="logins")
# Researcher table
class Researcher(db.Model):
    __tablename__ = 'researcher'
    user_id = Column(Integer, ForeignKey('user.user_id'),primary_key=True)
    institute = Column(String(100))
    user = relationship("User", backref="researchers")
```

```
# Advertisement table
class Advertisement(db.Model):
    __tablename__ = 'advertisement'
    ad_id = Column(Integer, primary_key=True)
    published_by = Column(String(100))
    type = Column(String(100))
    title = Column(String(100))
    category = Column(String(100))
    description = Column(String(100))
    date = Column(Date)
    time = Column(String(100))
    user_id = Column(Integer, ForeignKey('user.user_id'))
    unit_price = Column(Integer)
    crop_id = Column(Integer)
    amount = Column(Integer)
    telephone_no = Column(String(100))
    verified_officer_id = Column(Integer)
    image_link = Column(String(100))
    user = relationship("User", backref="advertisements")
# Farm table
class Farm(db.Model):
    __tablename__ = 'farm'
    farm_id = Column(Integer, primary_key=True)
    farm_name = Column(String(100))
    farmer_id = Column(Integer, ForeignKey('farmer.user_id'))
    address = Column(String(100))
    type = Column(String(100))
    area_of_field = Column(String(100))
    owner_nic = Column(String(100))
    owner_name = Column(String(100))
    recorded_by = Column(Integer)
    office_id = Column(Integer, ForeignKey('agri_office.agri_office_id'))
```

```
    field_area_id = Column(Integer, ForeignKey('field_area.field_area_id'))
    farmer = relationship("Farmer", backref="farms")
# Crop table
class Crop(db.Model):
    __tablename__ = 'crop'
    crop_id = Column(Integer, primary_key=True)
    crop_name = Column(String(100))
    breed = Column(String(100))
    description = Column(String(100))
    updated_by = Column(Integer)
    added_by = Column(Integer)
    added_date = Column(Date)
# DisasterInfo table
class DisasterInfo(db.Model):
    __tablename__ = 'disaster_info'
    disaster_info_id = Column(Integer, primary_key=True)
    cultivation_info_id = Column(Integer
```

# Appendix C

All the related schemas are as follows

```
# Farm schema

class FarmSchema(ma.Schema):

    class Meta:

        fields = ('farm_id', 'farm_name', 'address', 'type', 'farmer_id',
'area_of_field', 'owner_nic', 'owner_name' )


# Contact schema

class ContactSchema(ma.Schema):

    class Meta:

        fields = ('contact_id', 'user_id', 'number', 'area_code')


# Address schema

class AddressSchema(ma.Schema):

    class Meta:

        fields = ('address_id', 'user_id', 'city', 'town', 'street',
'home_no', 'home_name')


# SuperAdmin schema

class SuperAdminSchema(ma.Schema):

    class Meta:

        fields = ('user_id', 'employee_id', 'role_type')


# RegionalAdmin schema

class RegionalAdminSchema(ma.Schema):

    class Meta:

        fields = ('user_id', 'employee_id', 'managed_by_employee_id',
'district', 'province', 'agri_office_id', 'service_start_date')


# AgricultureOfficer schema

class AgricultureOfficerSchema(ma.Schema):
```

```
    class Meta:
        fields = ('user_id', 'employee_id', 'managed_by_employee_id',
'agri_office_id', 'service_start_date', 'field_area_id')


# AgriOffice schema
class AgriOfficeSchema(ma.Schema):
    class Meta:
        fields = ('agri_office_id', 'name', 'city', 'province', 'district')


# FieldArea schema
class FieldAreaSchema(ma.Schema):
    class Meta:
        fields = ('field_area_id', 'agri_office_id', 'name')


# Reports schema
class ReportsSchema(ma.Schema):
    class Meta:
        fields = ('report_id', 'category', 'date', 'time', 'user_id', 'link')


# Farmer schema
class FarmerSchema(ma.Schema):
    class Meta:
        fields = ('user_id', 'assigned_office_id', 'assigned_field_area_id',
'updated_by', 'added_by', 'registered_date', 'tax_file_no')


# Login schema
class LoginSchema(ma.Schema):
    class Meta:
        fields = ('user_id', 'username', 'encoded_pw')


# Vendor schema
class VendorSchema(ma.Schema):
    class Meta:
```

```
        fields = ('user_id', 'business_reg_no', 'tax_file_no')


# Researcher schema

class ResearcherSchema(ma.Schema):

    class Meta:

        fields = ('user_id', 'institute')


# Advertisement schema

class AdvertisementSchema(ma.Schema):

    class Meta:

        fields = ('ad_id', 'published_by', 'type', 'title', 'category',
'description', 'date', 'time', 'user_id', 'unit_price', 'crop_id', 'amount',
'telephone_no', 'verified_officer_id', 'image_link')


# Crop schema

class CropSchema(ma.Schema):

    class Meta:

        fields = ('crop_id', 'crop_name', 'breed', 'description',
'updated_by', 'added_by')


# DisasterInfo schema

class DisasterInfoSchema(ma.Schema):

    class Meta:

        fields = ('disaster_info_id', 'cultivation_info_id', 'date', 'time',
'damaged_area', 'estimated_damaged_harvest',
'estimated_damaged_harvest_value', 'type')


# Aid schema

class AidSchema(ma.Schema):

    class Meta:

        fields = ('aid_id', 'aid_name', 'aid_batch', 'year',
'in_charged_office_id', 'description')


# Fertilizer schema
```

```
class FertilizerSchema(ma.Schema):

    class Meta:

        fields = ('fertilizer_id', 'aid_id', 'manufactured_date', 'brand',
'batch_no', 'expiry_date', 'name', 'type', 'description')


# Pesticides schema

class PesticidesSchema(ma.Schema):

    class Meta:

        fields = ('pesticides_id', 'aid_id', 'manufactured_date', 'brand',
'batch_no', 'expiry_date', 'name', 'type', 'description')


# MonetaryAid schema

class MonetaryAidSchema(ma.Schema):

    class Meta:

        fields = ('monetaryAid_id', 'aid_id', 'description', 'reason')


# Fuel schema

class FuelSchema(ma.Schema):

    class Meta:

        fields = ('fuelAid_id', 'aid_id', 'reason', 'description',
'fuel_type')


# MiscellaneousAids schema

class MiscellaneousAidsSchema(ma.Schema):

    class Meta:

        fields = ('miscellaneousAids_id', 'aid_id', 'type', 'reason',
'description')


# AidDistribution schema

class AidDistributionSchema(ma.Schema):

    class Meta:

        fields = ('distribution_id', 'aid_id', 'agri_office_id', 'date',
'time', 'in_charged_officer_id', 'cultivation_info_id', 'farmer_id',
'amount_received', 'amount_approved', 'description')
```

# Appendix D

### Mobile Application Code

- Login page

```
// Import necessary packages
import 'package:flutter/material.dart';
import 'package:fluttertoast/fluttertoast.dart';
import 'package:http/http.dart' as http;// Import http package for making
HTTP requests
import 'dart:convert';// Import 'dart:convert' for JSON decoding//provides
encoders and decoders for converting between JSON and Dart objects.
import '../Home/HomePage.dart';
import 'RegisterAgriOfficer.dart';

// Define variables to keep user information who logged in the app
String? token;
String? firstname;
String? lastname;
String? email;
int? userid;

// Define the LoginPage widget
class LoginPage extends StatefulWidget {
 const LoginPage({super.key});

 @override
 State<LoginPage> createState() => _LoginPageState();
}

// Define the state for the LoginPage
class _LoginPageState extends State<LoginPage> {

 // Create controllers for username and password text fields
 TextEditingController username = TextEditingController();
 TextEditingController password = TextEditingController();

 // Define a method for login functionality
 Future<void> _login() async {
   const String apiUrl =
       'https://bluebird-balanced-drum.ngrok-free.app/user/login'; // API
Url:Login

   // Send a POST request to the login API
   final response = await http.post(
     Uri.parse(apiUrl),
     headers: <String, String>{
       'Content-Type': 'application/json; charset=UTF-8',
     },
     body: jsonEncode(<String, String>{
       'email': username.text,
       'password': password.text,
```

```
  }),
);

if (!mounted) return;

// Check the response status code
if (response.statusCode == 200) {
  // Parse the JSON response
  final Map<String, dynamic> responseData = jsonDecode(response.body);
  // Extract the role value from the JSON response
  final int role = responseData['role'];
  token=responseData['token'];
  firstname=responseData['firstname'];
  lastname=responseData['lastname'];
  email=responseData['email'];
  userid=responseData['user_id'];


  // Check the user role
  if (role == 4) {
    // Redirect to the home page if role is 4
  Navigator.pushReplacement(
    context,
    MaterialPageRoute(builder: (context) => const HomePage()),
  );
    // Show invalid login message if role is not 4
  } else if(role!=4){
    Fluttertoast.showToast(
      msg: 'Invalid login',
      toastLength: Toast.LENGTH_LONG,
      gravity: ToastGravity.BOTTOM,
      timeInSecForIosWeb: 1,
      backgroundColor: Colors.transparent,
      textColor: Colors.red,
      fontSize: 16.0,
    );
  }
  }else{
  // Show error message for failed login
  showDialog(
    context: context,
    builder: (BuildContext context) {
      return AlertDialog(
        title: const Text('Login Failed'),
        content: const Text('Invalid username or password.'),
        actions: <Widget>[
          TextButton(
            onPressed: () {
              Navigator.of(context).pop();
            },
            child: const Text('OK'),
          ),
        ],
```

```
        );
      },
    );
  }
}

// Define a method for navigating to the registration page
void performRegistration(BuildContext context) {
  Navigator.push(
    context,
    MaterialPageRoute(builder: (context) => const RegisterOfficer()),
  );

}

// Build the login page UI
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Center(
      child: SingleChildScrollView(
        child: Padding(
          padding: const EdgeInsets.all(30.0),
          child: Column(
            crossAxisAlignment: CrossAxisAlignment.center,
            children: <Widget>[
              Image.asset(
                "lib/assets/logo.png",
                width: 150.0,
                height: 150.0,
              ),
              const SizedBox(height: 30),
              TextFormField(
                controller: username,
                decoration: const InputDecoration(
                  labelText: "Username",
                  prefixIcon: Icon(Icons.person),
                ),
              ),
              const SizedBox(height: 30),
              TextFormField(
                controller: password,
                obscureText: true,
                decoration: const InputDecoration(
                  labelText: "Password",
                  prefixIcon: Icon(Icons.lock),
                ),
              ),
              const SizedBox(height: 30),
              ElevatedButton(
                onPressed:
                    _login, // Call the _login method when the button is
pressed
```

```
                child: const Text('Login'),
              ),
              const SizedBox(height: 20),
              Row(
                mainAxisAlignment: MainAxisAlignment.center,
                children: <Widget>[
                  Expanded(
                    child: Divider(
                      color: Colors.teal.shade300,
                      height: 10,
                    ),
                  ),
                  const Text("or"),
                  Expanded(
                    child: Divider(
                      color: Colors.teal.shade300,
                      height: 10,
                    ),
                  ),
                ],
              ),
              const SizedBox(height: 20),
              Row(
                mainAxisAlignment: MainAxisAlignment.center,
                children: <Widget>[
                  const Text(
                    'Don\'t have an account? ',
                    style: TextStyle(fontSize: 16),
                  ),
                  GestureDetector(
                    onTap: () {
                      performRegistration(context);
                    },
                    child: const Text(
                      "Register",
                      style: TextStyle(fontSize: 16, color: Colors.teal),
                    ),
                  )
                ],
              )
            ],
          ),
        ),
      ),
    );
  }
}
```

- Page for adding disaster data

```
import 'package:flutter/material.dart';
import 'package:fluttertoast/fluttertoast.dart';
import 'package:intl/intl.dart';
import 'package:myapp/Screens/Cultivation/SearchCultivation.dart';
import 'package:http/http.dart' as http;
import '../AgriOfficer/LoginPage.dart';
import '../Home/HomePage.dart';

class AddDisasterRecords extends StatefulWidget {
 const AddDisasterRecords({super.key});

 @override
 State<AddDisasterRecords> createState() => _AddDisasterRecordsState();
}

class _AddDisasterRecordsState extends State<AddDisasterRecords> {
 String? _selectedIssueValue;
 TextEditingController cultivationInfoId = TextEditingController();
 TextEditingController damagedArea = TextEditingController();
 TextEditingController damagedHarvestExtent = TextEditingController();
 TextEditingController estimatedLoss = TextEditingController();
 TextEditingController date = TextEditingController();

 Future<void> _performAddRecord() async{
   const String apiUrl =
       'https://bluebird-balanced-drum.ngrok-free.app/disaster/info';

   final response = await http.post(
     Uri.parse(apiUrl),
     headers: <String, String>{
       'Content-Type': 'application/json; charset=UTF-8',
       'Authorization': 'Bearer $token',
     },
     body: jsonEncode(<String, dynamic>{
       'cultivation_info_id': int.parse(cultivationInfoId.text),
       'damaged_area': int.parse(damagedArea.text),
       'estimated_damaged_harvest': damagedHarvestExtent.text,
       'estimated_damaged_harvest_value': estimatedLoss.text,
       'type': _selectedIssueValue,
       'date': date.text,

     }),
   );

   // print('Response body: ${response.body}');

   if (!mounted) return;
```

```
  if (response.statusCode == 200) {
    Navigator.pushReplacement(
      context,
      MaterialPageRoute(builder: (context) => const HomePage()),
    );
    Fluttertoast.showToast(
      msg: "Successfully added a new disaster record",
      toastLength: Toast.LENGTH_LONG,
      gravity: ToastGravity.BOTTOM,
      backgroundColor: Colors.black12,
      textColor: Colors.green,
      fontSize: 16.0,
    );
    // print(response.body
  } else if (response.statusCode == 400) {
    showDialog(
      context: context,
      builder: (BuildContext context) {
        return AlertDialog(
          title: const Text('Failed to Add Record'),
          content: const Text('Invalid Cultivation Info Id '),
          actions: <Widget>[
            TextButton(
              onPressed: () {
                Navigator.of(context).pop();
              },
              child: const Text('OK'),
            ),
          ],
        );
      },
    );
  } else {
    Fluttertoast.showToast(
      msg: "System Error! \nPlease Login & Try Again",
      toastLength: Toast.LENGTH_LONG,
      gravity: ToastGravity.BOTTOM,
      backgroundColor: Colors.black12,
      textColor: Colors.green,
      fontSize: 16.0,
    );
  }
 }



 void _performSearchCultivationInfoId(BuildContext context) {
   Navigator.push(
     context,
     MaterialPageRoute(builder: (context) => const
SearchCultivationPage()),
   );
 }
```

```
 Future<void> _selectDate(BuildContext context) async {
   final DateTime? picked = await showDatePicker(
     context: context,
     initialDate: DateTime.now(),
     firstDate: DateTime(2023),
     lastDate: DateTime.now(),
   );
   if (picked != null) {
     setState(() {
       // print(DateFormat('yyyy-MM-dd').format(picked));
       date.text = DateFormat('yyyy-MM-dd').format(picked);
     });
   }
 }
 List<DropdownMenuItem<String>> get dropdownItems {
   List<DropdownMenuItem<String>> menuItems = [
     DropdownMenuItem(
       child: Row(
         children: <Widget>[
           Icon(Icons.warning, color: Colors.orange), // Icon for Flood
           SizedBox(width: 10), // Add some space between icon and text
           Text("Flood"),
         ],
       ),
       value: "Flood",
     ),
     DropdownMenuItem(
       child: Row(
         children: <Widget>[
           Icon(Icons.cloud, color: Colors.blue), // Icon for Drought
           SizedBox(width: 10), // Add some space between icon and text
           Text("Drought"),
         ],
       ),
       value: "Drought",
     ),
     // Add similar DropdownMenuItem entries for other options
     DropdownMenuItem(
       child: Row(
         children: <Widget>[
           Icon(Icons.bug_report, color: Colors.green), // Icon for Pests &
Disease Outbreak
           SizedBox(width: 10), // Add some space between icon and text
           Text("Pests & Disease Outbreak"),
         ],
       ),
       value: "Pests & Disease Outbreak",
     ),
     // Add similar DropdownMenuItem entries for other options
     DropdownMenuItem(
       child: Row(
         children: <Widget>[
```

```
            Icon(Icons.storm, color: Colors.grey), // Icon for Storm
            SizedBox(width: 10), // Add some space between icon and text
            Text("Storm"),
          ],
        ),
        value: "Storm",
      ),
      // Add similar DropdownMenuItem entries for other options
      DropdownMenuItem(
        child: Row(
          children: <Widget>[
            Icon(Icons.warning, color: Colors.red), // Icon for Chemical
Spills
            SizedBox(width: 10), // Add some space between icon and text
            Text("Chemical Spills"),
          ],
        ),
        value: "Chemical Spills",
      ),
      // Add similar DropdownMenuItem entries for other options
      DropdownMenuItem(
        child: Row(
          children: <Widget>[
            Icon(Icons.landscape, color: Colors.brown), // Icon for Land
Degradation
            SizedBox(width: 10), // Add some space between icon and text
            Text("Land Degradation"),
          ],
        ),
        value: "Land Degradation",
      ),
      // Add similar DropdownMenuItem entries for other options
    ];
    return menuItems;
  }


  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text(
          "Add Disaster Records",
          style: TextStyle(fontSize: 30.0),
        ),
        backgroundColor: Colors.teal.shade200,
        leading: IconButton(
          icon: const Icon(Icons.home),
          onPressed: () {
            Navigator.pushReplacement(
              context,
              MaterialPageRoute(builder: (context) => const HomePage()),
            );
```

```
        },
      ),
    ),

    body: SingleChildScrollView(
      child: Padding(
        padding: const EdgeInsets.all(20.0),
        child: Column(

          crossAxisAlignment: CrossAxisAlignment.center,
          children: [
            Row(
              mainAxisAlignment: MainAxisAlignment.end,
              children: [
                const SizedBox(height: 100.0,),
                ElevatedButton.icon(
                    icon: const Icon(Icons.search,color: Colors.green,),
                    label: const Text(
                      "Cultivation Info Record Id",
                      style: TextStyle(fontSize: 18.0),
                    ),
                    onPressed: () {
                      _performSearchCultivationInfoId(context);
                    }),
              ],
            ),
            TextFormField(
              controller: cultivationInfoId,
              keyboardType:
                  const TextInputType.numberWithOptions(decimal: true),
              decoration: const InputDecoration(
                  labelText: "Cultivation Information Record Id",
                  prefixIcon:
Icon(Icons.indeterminate_check_box_outlined,color: Colors.teal,),
                  hintText: "Eg:03",
                  hintStyle: TextStyle(color: Colors.black12)),
              autovalidateMode: AutovalidateMode.onUserInteraction,
              validator: (value) {
                if (value == null || value.isEmpty) {
                  return 'Please Fill';
                }
                return null;
              },
            ),
            const SizedBox(
              height: 40.0,
            ),
            Row(
              children: [
                const Text(
                  "Disaster Type",
                  style: TextStyle(
                      fontSize: 18.0,
```

```
            fontWeight: FontWeight.bold,
            color: Colors.black),
      ),
      const SizedBox(width: 30,),
      DropdownButton<String>(
        value: _selectedIssueValue,
        items: dropdownItems,
        onChanged: (String? choice) {
          setState(() {
            _selectedIssueValue = choice;
          });
        },
        hint: const Text("------------Select------------"),
      ),
    ],
  ),
  const SizedBox(height:30, ),

  const Row(
    mainAxisAlignment: MainAxisAlignment
        .start, // Aligns widgets to the start of the main axis
    children: [
      Text(
        "Detailed Damage Assessment",
        style: TextStyle(
            fontSize: 18.0,
            fontWeight: FontWeight.bold,
            color: Colors.black),
      ),
    ],
  ),
  TextFormField(
    controller: damagedArea,
    keyboardType:
    const TextInputType.numberWithOptions(decimal: true),
    decoration: const InputDecoration(
        labelText: "Damaged Area in Acre [Estimated]",
        prefixIcon: Icon(Icons.landscape,color: Colors.teal,),
        hintText: "Eg: 1.5 Acre",
        hintStyle: TextStyle(color: Colors.black12)),
    autovalidateMode: AutovalidateMode.onUserInteraction,
    validator: (value) {
      if (value == null || value.isEmpty) {
        return 'Please Fill';
      }
      return null;
    },
  ),

  TextFormField(
    controller: damagedHarvestExtent,
    keyboardType:
    const TextInputType.numberWithOptions(decimal: true),
```

```
              decoration: const InputDecoration(
                  labelText: "Harvest Damage Extent [Estimation]",
                  prefixIcon: Icon(Icons.eco,color: Colors.teal,),
                  hintText: "Eg:340kg",
                  hintStyle: TextStyle(color: Colors.black12)),
              autovalidateMode: AutovalidateMode.onUserInteraction,
              validator: (value) {
                if (value == null || value.isEmpty) {
                  return 'Please Fill';
                }
                return null;
              },
            ),
            TextFormField(
              controller: estimatedLoss,
              keyboardType:
                  const TextInputType.numberWithOptions(decimal: true),
              decoration: const InputDecoration(
                  labelText: "Estimated Loss",
                  prefixIcon: Icon(Icons.money,color: Colors.teal,),
                  hintText: "Eg:in Lkr",
                  hintStyle: TextStyle(color: Colors.black12)),
              autovalidateMode: AutovalidateMode.onUserInteraction,
              validator: (value) {
                if (value == null || value.isEmpty) {
                  return 'Please Fill';
                }
                return null;
              },
            ),

            const SizedBox(width: 50),
            TextFormField(
              readOnly: true,
              controller: date,
              decoration: InputDecoration(
                labelText: "Date",
                prefixIcon: const Icon(Icons.calendar_today,color:
Colors.teal,),
                suffixIcon: IconButton(
                  icon: const Icon(Icons.edit_calendar_rounded),
                  onPressed: () => _selectDate(context),
                ),
              ),
            ),
            const SizedBox(height: 30.0,),
            ElevatedButton(
              onPressed: () {
                _performAddRecord();
              },
              child: const Text(
                'Submit',
                style: TextStyle(fontSize: 25.0),
```

- Code for get geo location

```
import 'package:flutter/material.dart';
import 'package:geocoding/geocoding.dart';
import 'package:geolocator/geolocator.dart';
import 'package:google_maps_flutter/google_maps_flutter.dart';

class GetCurrentLocation extends StatefulWidget {
 const GetCurrentLocation({Key? key}) : super(key: key);

 @override
 State<GetCurrentLocation> createState() => _GetCurrentLocationState();
}

class _GetCurrentLocationState extends State<GetCurrentLocation> {
 String? _currentAddress;
 Position? _currentPosition;
 GoogleMapController? _mapController;

 void _onMapCreated(GoogleMapController controller) {
   _mapController = controller;
 }

 Future<void> _getCurrentPosition() async {
   final hasPermission = await _handleLocationPermission();

   if (!hasPermission) return;
   await Geolocator.getCurrentPosition(desiredAccuracy:
LocationAccuracy.high)
       .then((Position position) {
     setState(() {
       _currentPosition = position;
       _initialCameraPosition = CameraPosition(
         target: LatLng(position.latitude, position.longitude),
         zoom: 14,
       );
       // Move map camera to the new location
       _mapController?.animateCamera(
           CameraUpdate.newCameraPosition(_initialCameraPosition));
       _getAddressFromLatLng(position);
     });
   }).catchError((e) {
     debugPrint(e);
   });
 }

 Future<bool> _handleLocationPermission() async {
   bool serviceEnabled;
   LocationPermission permission;

   serviceEnabled = await Geolocator.isLocationServiceEnabled();
   if (!serviceEnabled) {
```

```
      ScaffoldMessenger.of(context).showSnackBar(const SnackBar(
          content: Text(
              'Location services are disabled. Please enable the
services')));
      return false;
    }
    permission = await Geolocator.checkPermission();
    if (permission == LocationPermission.denied) {
      permission = await Geolocator.requestPermission();
      if (permission == LocationPermission.denied) {
        ScaffoldMessenger.of(context).showSnackBar(
            const SnackBar(content: Text('Location permissions are
denied')));
        return false;
      }
    }
    if (permission == LocationPermission.deniedForever) {
      ScaffoldMessenger.of(context).showSnackBar(const SnackBar(
          content: Text(
              'Location permissions are permanently denied, we cannot
request permissions.')));
      return false;
    }
    return true;
  }

  Future<void> _getAddressFromLatLng(Position position) async {
    await placemarkFromCoordinates(
        _currentPosition!.latitude, _currentPosition!.longitude)
        .then((List<Placemark> placemarks) {
      Placemark place = placemarks[0];
      setState(() {
        _currentAddress =
        '${place.street}, ${place.subLocality},
${place.subAdministrativeArea}, ${place.postalCode}';
      });
    }).catchError((e) {
      debugPrint(e);
    });
  }

  @override
  Widget build(BuildContext context) {
    print("in location page _");

    return Scaffold(
      appBar: AppBar(title: const Text("Location Page")),
      body: SafeArea(
        child: Column(
          children: [
            Expanded(
              child: GoogleMap(
                onMapCreated: _onMapCreated,
```

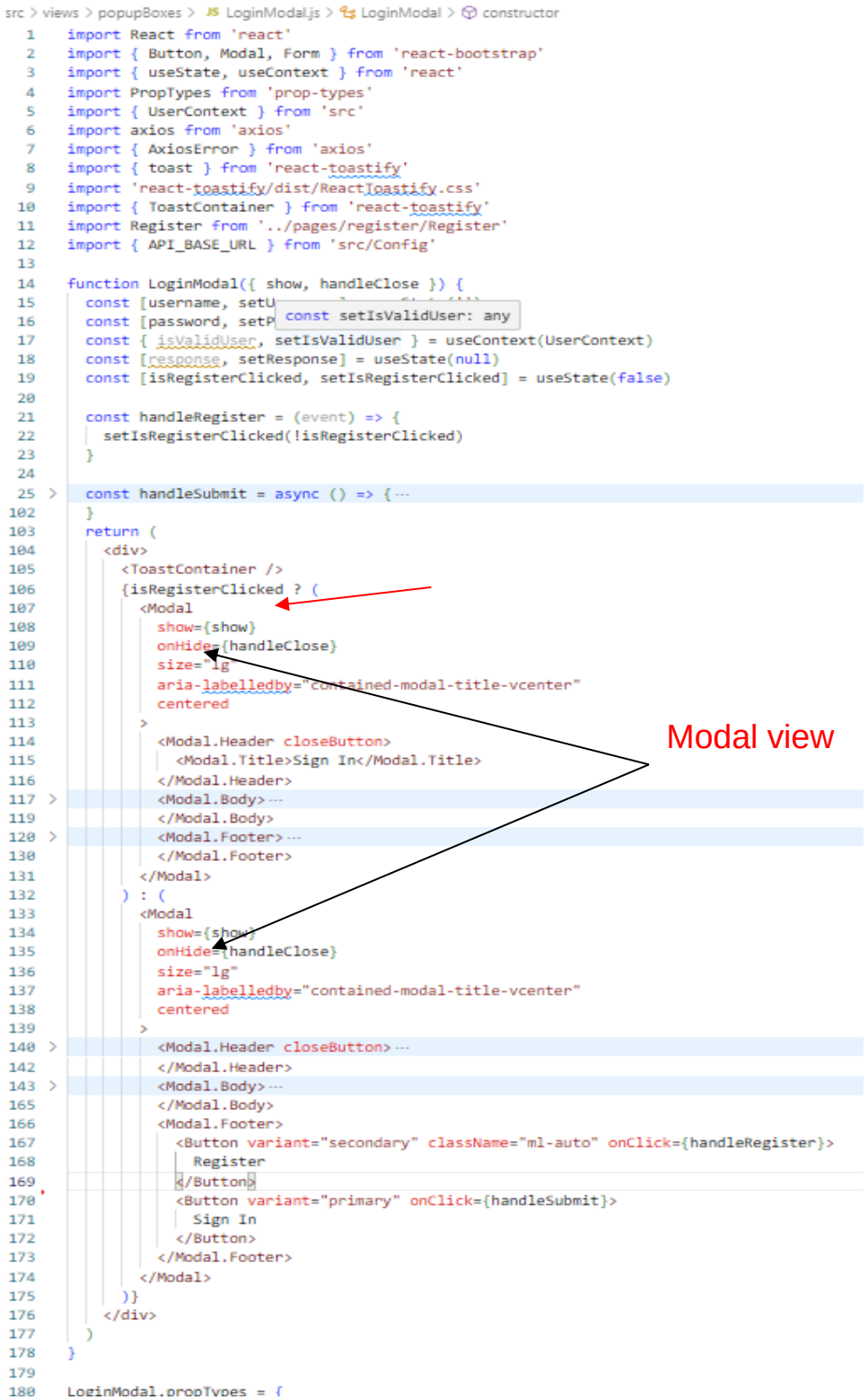
```
              initialCameraPosition: _initialCameraPosition,
              markers: {
                if (_currentPosition != null)
                  Marker(
                    markerId: const MarkerId("currentLocation"),
                    position: LatLng(
                      _currentPosition!.latitude,
                      _currentPosition!.longitude,
                    ),
                  ),
              },
            ),
          ),
          const SizedBox(height: 20),
          Text('LAT: ${_currentPosition?.latitude ?? ""}'),
          Text('LNG: ${_currentPosition?.longitude ?? ""}'),
          const SizedBox(height: 20),
          Text('ADDRESS: ${_currentAddress ?? ""}'),
          const SizedBox(height: 20),
          Row(crossAxisAlignment: CrossAxisAlignment.values[2], children:
[
            ElevatedButton(
              onPressed: _getCurrentPosition,
              child: const Text("Get Current Location"),
            ),
            ElevatedButton(
              onPressed: () {
                if (_currentPosition != null) {
                  // Return the latitude and longitude back to the
previous screen
                  Navigator.pop(context, {
                    'latitude': _currentPosition!.latitude,
                    'longitude': _currentPosition!.longitude,
                  });
                }
              },
              child: const Text("Confirm Location"),
            ),
          ])
        ],
      ),
    ),
```

# Appendix E

**Frontend implementation - React Web Application**

**Login component:**

The following code - Login component  is a reactjs function component that renders as a modal when the sign in button is clicked.



Landing page component:

The following code shows the landing page of the Web App

```
import { DefaultReportSet, AdminReportSet } from 'src/views/reports/ReportSet'
import Contact from 'src/components/landingpage/Contact'
import {
  DataAdminCollection,
  DataGenericCollection,
  DataOfficerCollection,
} from 'src/views/pages/dataCollection/DataCollection'
import AdminReport from '../reports/AdminReport'
import OfficerReport from '../reports/OfficerReport'
import ProductListPage from 'src/views/marketplace/forms/ProductListPage'

function LandingPage() {
  const { isValidUser, setIsValidUser } = useContext(UserContext)
  const [selectedNavItem, setSelectedNavItem] = useState('Home')
  const [user, setUser] = useState(null)
  const [userRole, setUserRole] = useState(null)
  const [isAdmin, setIsAdmin] = useState(false)
  const [isOfficer, setOIsOfficer] = useState(false)

  //UseEffect to check if the user is valid
  useEffect(() => {
    if (isValidUser) {
      const user = JSON.parse(localStorage.getItem('user')) //Get the user from the local storage
      if (!user) {
        setIsValidUser(false) //If the user is not valid, set the valid user to false
      }
      setUser(user)
      setUserRole(user.role)
      if (user.role === 1) {
        setIsAdmin(true)
      } else if (user.role === 4) {
        setOIsOfficer(true)
      }
    }
  }, [isValidUser, setIsValidUser]) //The useEffect will run when the isValidUser changes

  //Handle the navigation bar click
  const handleContent = (navItem) => {
    setSelectedNavItem(navItem)
  }

  // returns the landing page view
  return (
    <div>
      <DefaultLayout2>
        <div className="App-header">
          <NavigationBar handleNavClick={handleContent} />
        </div>
        {selectedNavItem === 'About' && <About />}
        {selectedNavItem === 'Home' && <MainContent />}
        {selectedNavItem === 'Latest_Reports' &&
          (isAdmin ? <AdminReport /> : isOfficer ? <OfficerReport /> : <DefaultReportSet />)}
        {selectedNavItem === 'Contact' && <Contact />}
        {selectedNavItem === 'DataCollection' && //Check the user role and display the relevant data collection page
          (isAdmin ? (
            <DataAdminCollection />
          ) : isOfficer ? (
            <DataOfficerCollection />
          ) : (
            <DataGenericCollection />
          ))}
        {selectedNavItem === 'DataOfficerCollection' && // Check the user role and display the relevant data collection page
          (isOfficer ? <DataOfficerCollection /> : <></>)}
        {selectedNavItem === 'Free Advertising Support' && <ProductListPage />}
      </DefaultLayout2>
    </div>
  )
}

// Export the LandingPage component
export default LandingPage
```

The Report View component for Administrator role

```
//Admin report viewer
const AdminReportViewer = () => {
  const [activeKey, setActiveKey] = useState('')

  //Handle the item click
  const handleItemClick = (key) => {
    setActiveKey(key)
  }

  return (
    <CCard>
      <CContainer>
        <CRow>
          <CCol>
            <CListGroup>
              <CListGroupItem className="blue-link" onClick={() => handleItemClick('User Reports')}>
                Total Registered Users
              </CListGroupItem>
              <CListGroupItem
                className="blue-link"
                onClick={() => handleItemClick('Estimated Harvest vs Actual')} //item for estimated harvest vs actual
              >
                Estimated Harvest vs Actual
              </CListGroupItem>
              <CListGroupItem···
              </CListGroupItem>
              <CListGroupItem···
              </CListGroupItem>
              <CListGroupItem···
              </CListGroupItem>
              <CListGroupItem···
              </CListGroupItem>
              <CListGroupItem
                className="blue-link"
                onClick={() => handleItemClick('Tax Payer Report')}
              >
                Tax Prayer Report
              </CListGroupItem>
              <CListGroupItem
                className="blue-link"
                onClick={() => handleItemClick('Disaster Overview')} //Change the key
              >
                Disaster Overview Report
              </CListGroupItem>
              <CListGroupItem
                className="blue-link"
                onClick={() => handleItemClick('AdvertisementServiceReport')} //Change the key
              >
                Advertisement Overview Report
              </CListGroupItem>
            </CListGroup>
          </CCol>
          <CCol className="viewer">
            {activeKey === 'User Reports' && <UsersGroupByRoleByAdmin />} // checks the active key and display the relevant report
            {activeKey === 'Estimated Harvest vs Actual' && <HarvestEstimatedVsActual />} // checks the active key and display the relevant report
            {activeKey === 'AidDistribution' && <AidDistributionByAidTypeAdmin />}
            {activeKey === 'New Aid distribution' && <AidFundingAdminTable />}
            {activeKey === 'Crop Yield Reports' && <LankaMapByCropYieldAdmin />}
            {activeKey === 'Field Mapping' && <LankaMapByFieldMapping />}
            {activeKey === 'Tax Payer Report' && <TotalTaxPayerReport />}
            {activeKey === 'Disaster Overview' && <AdminDisasterOverview />}
            {activeKey === 'AdvertisementServiceReport' && <AdminAdvertisementServiceReport />}
          </CCol>
        </CRow>
      </CContainer>
    </CCard>
```

The following page shows the component code for navigation bar, where the role based component is mainly handled in the web app.

```
//Navigation bar component
function NavigationBar({ handleNavClick }) {
  const { isValidUser, setIsValidUser } = useContext(UserContext)
  const [username, setUsername] = useState('')
  const [role, setRole] = useState(0)

  useEffect(() => {
    if (!isValidUser) {
      console.log(isValidUser)
      localStorage.clear()
    }
  }, [isValidUser])

  const [show, setShow] = useState(false)

  const handleSignIn = () => {
    console.log('Sign In')
    setShow(true)
    handleNavClick('Home')
  }

  // Handle the sign out
  const handleSignOut = () => {
    setIsValidUser(false)
    localStorage.clear()
    setRole(0)
    handleNavClick('Home')
  }

  //UseEffect to check if the user is valid
  useEffect(() => {
    if (isValidUser) {
      const user = JSON.parse(localStorage.getItem('user')) //Get the user from the local storage
      if (user && user.first_name) {
        setUsername(user.first_name) //Set the username
        setRole(user.role)
        console.log(user)
      }
    } else {
      handleSignOut()
    }
  }, [isValidUser]) //The useEffect will run when the isValidUser changes

  return (
    <div className="NavigationBar">
      <Navbar expand="lg" className="bg-body-tertiary">
        <Navbar.Brand href="#home" style={{ marginRight: '100px' }}>
          {' '}
          <Image src={Logo} rounded width={100} height={50} />
        </Navbar.Brand>
        <Navbar.Toggle aria-controls="basic-navbar-nav" />
        <Navbar.Collapse id="basic-navbar-nav">
          <Nav className="me-auto">
            <Nav.Link style={{ fontSize: '30px' }} onClick={() => handleNavClick('Home')}>
              Home
            </Nav.Link>
              
            <NavDropdown title="Services" id="basic-nav-dropdown" style={{ fontSize: '30px' }}>
              <NavDropdown.Item
                onClick={() => //Role based navigation
                  role === 1
                    ? handleNavClick('DataCollection')
                    : role === 4
                    ? handleNavClick('DataOfficerCollection')
                    : role === 5
                    ? handleNavClick('DataCollection')
                    : null
```

Show-map component that used to view map on the  given coordinations

```jsx
import React, { useState } from 'react'
import { MapContainer, TileLayer, Marker } from 'react-leaflet'
import 'leaflet/dist/leaflet.css'
import { Icon } from 'leaflet'

const ShowMap = () => {
  const defaultLongitude = parseFloat(localStorage.getItem('longitude'))
  const defaultLatitude = parseFloat(localStorage.getItem('latitude'))
  const coordinates = [defaultLatitude, defaultLongitude]

  console.log('Longitude from local storage:', defaultLongitude)
  console.log('Latitude from local storage:', defaultLatitude)

  const customIcon = new Icon({
    iconUrl: 'https://cdn-icons-png.flaticon.com/128/8326/8326599.png',
    iconSize: [38, 38],
  })

  return (
    <div>
      <h1>Select Map Area</h1>
      <MapContainer center={coordinates} zoom={30} style={{ height: '400px', width: '100%' }}>
        <TileLayer url="https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png" />
        <Marker position={coordinates} icon={customIcon} />
      </MapContainer>
    </div>
  )
}

export default ShowMap
```

- GeoJson used in showing Interactive Choropleth Map

- GeoJSON is a format for encoding geographical data structures. It's commonly used to represent spatial data like points, lines, and polygons, along with their attributes, in a simple and human-readable way. Essentially, it's a way to store and exchange geographic information in a format that computers can understand easily.

https://geojson.org/

- GeoJson for Sri Lanka:  https://github.com/MalakaGu/Sri-lanka-maps/tree/master/discrict_map

- Using GeoJson for sri lanka,  choropleth map was created for view island wide agricultural informations

```
// for each feature in the geojson, add mouseover and mouseout event handlers
function onEachFeature(feature, layer) {
  layer.on({
    mouseover: function (e) {
      const layer = e.target

      layer.setStyle({
        weight: 5,
        color: '#666',
        dashArray: '',
        fillOpacity: 0.9,
      })

      if (!L.Browser.ie && !L.Browser.opera && !L.Browser.edge) {
        layer.bringToFront()
      }
            (property) React.MutableRefObject<any>.current: any
      info.current.update(layer.feature.properties) // pass the properties of the hovered layer
    },
    mouseout: function (e) {
      geojson.resetStyle(e.target)
      info.current.update()
    },
  })
}

// Add the geojson to the map
if (data) {
  geojson = L.geoJSON(data, {
    style: function (feature) {
      return { // style each feature
        fillColor: getColor(feature.properties.total_harvested),
        weight: 2,
        opacity: 1,
        color: 'white',
        dashArray: '3',
        fillOpacity: 0.5,
      }
    },
    onEachFeature: onEachFeature, // add the event handlers to each feature
  }).addTo(map)
}
}, [map, data]) // re-run this effect when the map or data changes
```

```
<MapContainer
  key={mapKey}
  center={center}
  zoom={8}
  style={{ height: '700px', width: '500px' }}
  dragging={false}
  touchZoom={false}
  doubleClickZoom={false}
  scrollWheelZoom={false}
  keyboard={false}
  zoomControl={false}
>
  <TileLayer
    url="https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.
    attribution='&copy; <a href="http://www.openstreetma
    maxZoom={8}
  />
  <MapFeatureDataLayer formData={formData} />
</MapContainer>
```

## Chart component used in the system

1. Pie chart

```jsx
import React from 'react'
import PropTypes from 'prop-types'
import { CChart } from '@coreui/react-chartjs'
import { CContainer } from '@coreui/react'

const getRandomColor = () => {
  const letters = '0123456789ABCDEF'
  let color = '#'
  for (let i = 0; i < 6; i++) {
    color += letters[Math.floor(Math.random() * 16)]
  }
  return color
}

const PieChart = ({ data }) => {
  const labels = Object.keys(data)
  const chartData = {
    labels: labels,
    datasets: [
      {
        data: Object.values(data),
        backgroundColor: labels.map(() => getRandomColor()),
        borderColor: labels.map(() => getRandomColor()),
        borderWidth: 1,
      },
    ],
  }

  return (
    <CContainer fluid>
      <CChart
        type="pie"
        data={chartData}
        options={{
          plugins: {
            legend: {
              labels: {
                color: '#000000',
              },
            },
          },
        }}
      />
    </CContainer>
  )
}

PieChart.propTypes = {
  data: PropTypes.object.isRequired,
}
export default PieChart
```

2. Bar chart

```
  return (
    <div>
      <CFormSelect custom name="year" id="year" onChange={handleYearChange}>
        <option value="2020">2020</option>
        <option value="2021">2021</option>
        <option value="2022">2022</option>
        <option value="2023">2023</option>
      </CFormSelect>
      <CChartBar
        style={{ height: '300px', marginTop: '40px' }} //Component for the bar chart
        data={{
          labels: harvestData.map((data) => data.crop_name), // The labels for the x-axis
          datasets: [ // The data for the y-axis
            {
              label: 'Harvest Amount',
              backgroundColor: 'green',
              data: harvestData.map((data) => data.total_harvested_amount),
            },
          ],
        }}
        options={{ // The options for the chart
          maintainAspectRatio: false,
          scales: {
            x: {
              grid: {
                drawOnChartArea: false,
              },
            },
            y: {
              ticks: { // The ticks for the y-axis
                beginAtZero: true,
                maxTicksLimit: 5,
                stepSize: Math.ceil(250 / 5),
                max: 250,
              },
            },
          },
        }}
      />
    </div>
  )
}
```

# Appendix F

### Backend code for Python Flask API

Init.py for flask application initialization with routes and database

app > __init__.py > ...

```python
from app.route.communication_routes import com_routes
from app.route.aid_routes import aid_routes
from app.route.marketplace_routes import market_routes
from app.route.disaster_routes import disaster_routes
import string

from flask_mail import Mail, Message
from datetime import datetime, timedelta

app = Flask(__name__)

# Set SQLite DB directory
basedir = os.path.abspath(os.path.dirname(__file__))
# Set Flask SQLAlchemy config of the DB file location
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://root@localhost/agriInfo'
# Configire JWT secret key
app.config['JWT_SECRET_KEY'] = 'super_key'


# Initialize SQLAlchemy and Marshmallow
db.init_app(app)
ma.init_app(app)
jwt = JWTManager(app)
configure_mail(app)

# Register the app_blueprint
app.register_blueprint(app_blueprint)
app.register_blueprint(user_routes, url_prefix='/user')
app.register_blueprint(farm_routes, url_prefix='/farm')
app.register_blueprint(crop_routes, url_prefix='/crop')
app.register_blueprint(cultivation_routes, url_prefix='/cultivation')
app.register_blueprint(aid_routes, url_prefix='/aid')
app.register_blueprint(report_routes, url_prefix='/report')
app.register_blueprint(com_routes, url_prefix='/communication')
app.register_blueprint(market_routes, url_prefix='/market')
app.register_blueprint(disaster_routes, url_prefix='/disaster')

if __name__ == '__main__':
    app.run()

// database seed
@app.cli.command('db_create')
def db_create():
    db.create_all()
    print("DB created!")

```

## Farmer routes

```
#############################################################################################
                                #    User:Farmer
#############################################################################################


def add_farmer_to_system(data, user_id):

    try:
        # Check if the current user is an admin (modify 'admin' to your actual admin role)
        current_user = User.query.filter_by(user_id=user_id).first()
        if current_user.role not in [1, 3, 4]:
            message="Unauthorized to access this resource"
            return False, message, {}
        else:
            # Create a new instance of the Farmer model with the data
            new_farmer = Farmer(**data)

            new_farmer.added_by = current_user.user_id
            new_farmer.updated_by = current_user.user_id
            new_farmer.registered_date = parse_date(datetime.datetime.now().strftime('%Y-%m-%d'))


            # Add the new farmer to the database session
            db.session.add(new_farmer)
            user = User.query.get(new_farmer.user_id)
            data = {'role': 5}
            Update_User(data, user)

            # Commit the changes to the database
            db.session.commit()
            message="Successfully added farmer"

            # Return the JSON representation of the new farmer
            return True, message, new_farmer

    except Exception as e:
        db.session.rollback()
        logging.error(e)
        message=f"Farmer registration failure, please check this user is already registered, Ping
        return False, message, {}
```

```
        user_id = get_jwt_identity()
        isSucceed, message, new_farmer=add_farmer_to_system(data, user_id)
        if isSucceed:
            return farmer_schema.jsonify(new_farmer),200
        elif not isSucceed and message=="Unauthorized to access this resource":
            jsonify(message=message), 403
        else:
            return jsonify(message=message), 500
```

# Appendix G

This code creates a screen to obtain the current location of the device using the Geolocator and Geocoding plugins. It displays a Google Map with the current location marker, latitude, longitude, and address. Users can fetch the current location with a button and confirm it to return the latitude and longitude to the previous screen.

```
import 'package:flutter/material.dart';
import 'package:geocoding/geocoding.dart';
import 'package:geolocator/geolocator.dart';
import 'package:google_maps_flutter/google_maps_flutter.dart';

class GetCurrentLocation extends StatefulWidget {
  const GetCurrentLocation({Key? key}) : super(key: key);

  @override
  State<GetCurrentLocation> createState() => _GetCurrentLocationState();
}

class _GetCurrentLocationState extends State<GetCurrentLocation> {
  String? _currentAddress;
  Position? _currentPosition;
  GoogleMapController? _mapController;

// Initial camera position
  late CameraPosition _initialCameraPosition = const CameraPosition(
    target: LatLng(
        0, 0), // Default to a neutral location before getting actual
location
    zoom: 14,
  );

  void _onMapCreated(GoogleMapController controller) {
    _mapController = controller;
  }

  Future<void> _getCurrentPosition() async {
    final hasPermission = await _handleLocationPermission();

    if (!hasPermission) return;
    await Geolocator.getCurrentPosition(desiredAccuracy:
LocationAccuracy.high)
        .then((Position position) {
      setState(() {
        _currentPosition = position;
        _initialCameraPosition = CameraPosition(
          target: LatLng(position.latitude, position.longitude),
          zoom: 14,
        );
        // Move map camera to the new location
```

```
        _mapController?.animateCamera(
            CameraUpdate.newCameraPosition(_initialCameraPosition));
        _getAddressFromLatLng(position);
      });
    }).catchError((e) {
      debugPrint(e);
    });
  }

  Future<bool> _handleLocationPermission() async {
    bool serviceEnabled;
    LocationPermission permission;

    serviceEnabled = await Geolocator.isLocationServiceEnabled();
    if (!serviceEnabled) {
      ScaffoldMessenger.of(context).showSnackBar(const SnackBar(
          content: Text(
              'Location services are disabled. Please enable the
services')));
      return false;
    }
    permission = await Geolocator.checkPermission();
    if (permission == LocationPermission.denied) {
      permission = await Geolocator.requestPermission();
      if (permission == LocationPermission.denied) {
        ScaffoldMessenger.of(context).showSnackBar(
            const SnackBar(content: Text('Location permissions are
denied')));
        return false;
      }
    }
    if (permission == LocationPermission.deniedForever) {
      ScaffoldMessenger.of(context).showSnackBar(const SnackBar(
          content: Text(
              'Location permissions are permanently denied, we cannot request
permissions.')));
      return false;
    }
    return true;
  }

  Future<void> _getAddressFromLatLng(Position position) async {
    await placemarkFromCoordinates(
        _currentPosition!.latitude, _currentPosition!.longitude)
        .then((List<Placemark> placemarks) {
      Placemark place = placemarks[0];
      setState(() {
        _currentAddress =
        '${place.street}, ${place.subLocality},
${place.subAdministrativeArea}, ${place.postalCode}';
      });
    }).catchError((e) {
      debugPrint(e);
```

```
    });
  }

  @override
  Widget build(BuildContext context) {
    print("in location page _");

    return Scaffold(
      appBar: AppBar(title: const Text("Location Page")),
      body: SafeArea(
        child: Column(
          children: [
            Expanded(
              child: GoogleMap(
                onMapCreated: _onMapCreated,
                initialCameraPosition: _initialCameraPosition,
                markers: {
                  if (_currentPosition != null)
                    Marker(
                      markerId: const MarkerId("currentLocation"),
                      position: LatLng(
                        _currentPosition!.latitude,
                        _currentPosition!.longitude,
                      ),
                    ),
                },
              ),
            ),
            const SizedBox(height: 20),
            Text('LAT: ${_currentPosition?.latitude ?? ""}'),
            Text('LNG: ${_currentPosition?.longitude ?? ""}'),
            const SizedBox(height: 20),
            Text('ADDRESS: ${_currentAddress ?? ""}'),
            const SizedBox(height: 20),
            Row(crossAxisAlignment: CrossAxisAlignment.values[2], children: [
              ElevatedButton(
                onPressed: _getCurrentPosition,
                child: const Text("Get Current Location"),
              ),
              ElevatedButton(
                onPressed: () {
                  if (_currentPosition != null) {
                    // Return the latitude and longitude back to the previous
screen
                    Navigator.pop(context, {
                      'latitude': _currentPosition!.latitude,
                      'longitude': _currentPosition!.longitude,
                    });
                  }
                },
                child: const Text("Confirm Location"),
              ])
```

# Appendix H

Unittest sample code of backend for “User” related functionalities.

```
from datetime import datetime, timedelta
import datetime
from app import app
import jwt
import os
import unittest
from unittest.mock import patch
from flask import Flask
from flask import current_app  # Use Flask's current_app for app context specific
configurations

from app.models import db, User
from    app.service.users.user_service    import    Check_User_Token_Expiration,
Get_User_Information,   Search_User,   Update_User,   Validate_User,   deleteUser,
get_access_token,   getUserBy_Email,   getUserBy_Id,   register_user,   user_login,
isExistingUser
from app.service.users.util_service import parse_date
from flask_jwt_extended import JWTManager
import xmlrunner
from app.route import user_routes


class TestUserRoutes(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls.app = Flask(__name__)
        cls.app.config['TESTING'] = True
        cls.app.config['JWT_SECRET_KEY'] = 'super_key'
        basedir = os.path.abspath(os.path.dirname(__file__))
                cls.app.config['SQLALCHEMY_DATABASE_URI']   =   'sqlite:///'   +
os.path.join(basedir, 'test_agriInfo.db')
        cls.app.register_blueprint(user_routes.user_routes)
        db.init_app(cls.app)

        with cls.app.app_context():
            db.create_all()

    @classmethod
    def tearDownClass(cls):
        with cls.app.app_context():
            db.session.remove()
            db.drop_all()

    def setUp(self):
        # Ensure each test has a clean database
        with self.app.app_context():
```

```
        db.session.query(User).delete()
        db.session.commit()

    def test_user_registration(self):
        with self.app.app_context():
            user = User(
                first_name='fName',
                middle_name='mName',
                last_name='lName',
                nic='testNIC',
                email='test@example.com',
                password='test',
                dob=parse_date('1990-01-01'),
                role=1,
            )
            isSuccess, message = register_user(user)
            self.assertTrue(isSuccess)
            self.assertEqual(message, 'Registration success!')

    def test_user_login(self):
        with self.app.app_context():
            user = User(
                first_name='Jane',
                middle_name='Doe',
                last_name='Smith',
                nic='987654321V',
                email='jane@example.com',
                password='password',
                dob=parse_date('1992-02-02'),
                role=1,
            )
            db.session.add(user)
            db.session.commit()

            login_user = User(email='jane@example.com', password='password')
            loggedInUser = user_login(login_user)
            self.assertEqual(loggedInUser.email, 'jane@example.com')

    def test_access_token_generation(self):
        with self.app.app_context():
            user = User.query.filter_by(email='test@example.com').first()
            if user:
                token = get_access_token(user)
                self.assertIsNotNone(token)

    def test_user_deletion(self):
        with self.app.app_context():
            user = User.query.filter_by(email='jane@example.com').first()
            if user:
                isDeleted, message, _ = deleteUser(user.id)
                self.assertTrue(isDeleted)
                self.assertEqual(message, "User successfully deleted.")
```

```
    def test_get_user_by_id(self):
        with self.app.app_context():
             # Setup: create two users, one as the 'current user' and another as
the target user
            current_user = User(
                user_id=1,
                email='current@example.com',
                password='password',
                role=1  # Assuming role 1 is allowed to fetch other users
            )
            target_user = User(
                user_id=2,
                email='target@example.com',
                password='password',
                role=2
            )
            db.session.add(current_user)
            db.session.add(target_user)
            db.session.commit()

            # Test: Attempt to get the target user by ID using the current user's
ID
            retrieved_user = getUserBy_Id(2, 1)

            # Verify: Check that the retrieved user is the target user
            self.assertIsNotNone(retrieved_user)
            self.assertEqual(retrieved_user.user_id, target_user.user_id)
            self.assertEqual(retrieved_user.email, target_user.email)

    def test_get_user_by_id_unauthorized_role(self):
        with self.app.app_context():
           # Setup: create two users, one as the 'current user' with an unauthorized
role and another as the target user
            current_user = User(
                user_id=3,
                email='unauthorized@example.com',
                password='password',
                role=5  # Assuming role 5 is not allowed to fetch other users
            )
            target_user = User(
                user_id=4,
                email='another_target@example.com',
                password='password',
                role=2
            )
            db.session.add(current_user)
            db.session.add(target_user)
            db.session.commit()

             # Test: Attempt to get the target user by ID using the unauthorized
current user's ID
```

```
            retrieved_user = getUserBy_Id(4, 3)

             # Verify: Check that the retrieved user is None or access is denied
based on your function logic
            self.assertIsNone(retrieved_user)

    def test_get_user_by_email_authorized(self):
        # Test retrieving a user by email with an authorized current user.
        with self.app.app_context():
           # Setup: create an authorized current user and another user to retrieve
            authorized_user = User(
                user_id=10,  # Ensure unique user_id
                email='authorized@example.com',
                password='password',
                role=1  # Assuming roles 1, 3, 4 are authorized
            )
            target_user = User(
                user_id=20,  # Ensure unique user_id
                email='targetuser@example.com',
                password='password',
                role=2
            )
            db.session.add(authorized_user)
            db.session.add(target_user)
            db.session.commit()

             # Act: Attempt to get the target user by email using the authorized
current user's ID
            retrieved_user = getUserBy_Email('targetuser@example.com', 10)

            # Assert: Verify that the correct user is retrieved
            self.assertIsNotNone(retrieved_user)
            self.assertEqual(retrieved_user.email, 'targetuser@example.com')

    def test_get_user_by_email_unauthorized(self):
        # Test retrieving a user by email with an unauthorized current user.
        with self.app.app_context():
              # Setup: create an unauthorized current user and another user to
retrieve
            unauthorized_user = User(
                user_id=30,  # Ensure unique user_id
                email='unauthorized@example.com',
                password='password',
                role=5  # Assuming this role is unauthorized
            )
            another_target_user = User(
                user_id=40,  # Ensure unique user_id
                email='another_target@example.com',
                password='password',
                role=2
            )
            db.session.add(unauthorized_user)
```

```
            db.session.add(another_target_user)
            db.session.commit()

            # Act: Attempt to get the target user by email using the unauthorized
current user's ID
            retrieved_user = getUserBy_Email('another_target@example.com', 30)

             # Assert: Verify that the user is not retrieved due to unauthorized
access
            self.assertIsNone(retrieved_user)

    def test_update_user(self):
        # Test updating user information.
        with self.app.app_context():
            # Setup: Create a user to update
            original_user = User(
                email='update@example.com',
                password='originalPassword',
                first_name='Original',
                last_name='User',
                nic='123456789V',
                dob=parse_date('1990-01-01'),
                role=1,
                middle_name='Middle'
            )
            db.session.add(original_user)
            db.session.commit()

            # Define the update data
            update_data = {
                'password': 'newPassword',
                'first_name': 'Updated',
                'last_name': 'User',
                'nic': '987654321V',
                'dob': '1995-05-05',
                'role': 2,
                'middle_name': 'UpdatedMiddle'
            }

            # Act: Call the Update_User function with the update data
            Update_User(update_data, original_user)

            # Fetch the updated user from the database
           updated_user = User.query.filter_by(email='update@example.com').first()

            # Assert: Verify the user's information has been updated
            self.assertEqual(updated_user.password, update_data['password'])
            self.assertEqual(updated_user.first_name, update_data['first_name'])
            self.assertEqual(updated_user.last_name, update_data['last_name'])
            self.assertEqual(updated_user.nic, update_data['nic'])
            self.assertEqual(updated_user.dob, parse_date(update_data['dob']))
            self.assertEqual(updated_user.role, update_data['role'])
```

```
            self.assertEqual(updated_user.middle_name, update_data['middle_name'])

    def test_search_user_with_filters(self):
        # Test searching users with specific filters.
        with self.app.app_context():
            # Setup: Add multiple users to test the filter and pagination
            users_to_add = [
                                User(email='user1@example.com',   first_name='Test',
last_name='User', role=1),
                                User(email='user2@example.com',   first_name='Test',
last_name='User2', role=2),
                            User(email='user3@example.com',  first_name='Another',
last_name='User3', role=1),
            ]
            for user in users_to_add:
                db.session.add(user)
            db.session.commit()

            # Define filters to search for users with a specific first name
            filters = {'first_name': 'Test', 'page': 1, 'per_page': 2}
            # Act: Search users using the filters
            result = Search_User(filters)
            # Assert: Check that the result matches expected structure and content
            self.assertEqual(result['page'], 1)
            self.assertEqual(result['per_page'], 2)
            self.assertTrue(result['total_pages'] >= 1)
            self.assertTrue(result['total_users'] >= 2)
            self.assertEqual(len(result['users']), 2)
                 self.assertTrue(all(user['first_name']  ==  'Test'  for  user  in
result['users']))

    def test_validate_user_success(self):
        # Test user validation succeeds when email and user_id match.
        with self.app.app_context():
            # Setup: Create a user to validate
            user = User(
                user_id=100,  # Ensure a unique user_id
                email='valid@example.com',
                password='password',
                role='1'
                )
            db.session.add(user)
            db.session.commit()
            # Act: Attempt to validate the created user by user_id and email
            is_valid, validated_user = Validate_User(100, 'valid@example.com')


# Assert: Verify that validation succeeds
            self.assertTrue(is_valid)
            self.assertIsNotNone(validated_user)
            self.assertEqual(validated_user.email, 'valid@example.com')
```

```
    def test_validate_user_failure(self):
        # Test user validation fails when email does not match user_id.
        with self.app.app_context():
            # Setup: Create a user to attempt to validate incorrectly
            user = User(
                user_id='101',  # Ensure a unique user_id
                email='invalid@example.com',
                password='password',
                role='1')
            db.session.add(user)
            db.session.commit()
           # Act: Attempt to validate the user with a correct user_id but incorrect
email
            is_valid, validated_user = Validate_User(101, 'wrong@example.com')
            # Assert: Verify that validation fails
            self.assertFalse(is_valid)
            self.assertIsNone(validated_user)

    def test_get_user_information(self):
        # Test retrieving information for a specific user.
        with self.app.app_context():
            # Setup: Create a user whose information will be retrieved
            new_user = User(
                user_id=123,  # Make sure this ID is unique or auto-generated
                email='info@example.com',
                password='securePassword',
                first_name='Test',
                last_name='User',
                role=2,  # Example role
                # Add any other required fields
            )
            db.session.add(new_user)
            db.session.commit()
            # Get the user's ID (if not manually set)
            user_id = new_user.user_id
            # Act: Retrieve the user information using the function under test
            retrieved_user = Get_User_Information(user_id)
             # Assert: Verify that the retrieved information matches the created
user        self.assertIsNotNone(retrieved_user)
            self.assertEqual(retrieved_user.email, 'info@example.com')
            self.assertEqual(retrieved_user.first_name, 'Test')
            self.assertEqual(retrieved_user.last_name, 'User')
            self.assertEqual(retrieved_user.role, 2)
            # Add any other assertions for fields you care about


if __name__ == '__main__':
     unittest.main(verbosity=2,  testRunner=xmlrunner.XMLTestRunner(output='test-
reports'))
```

# Appendix I

Testing done with Postman are represented below for your reference.

| Test Case | EndPoint | Method Type | Integration Test Pass/ Fail |
|---|---|---|---|
| RegisterUser | http://127.0.0.1:5000/user/register | POST | Pass |
| UserLogin | http://127.0.0.1:5000/user/login | POST | Pass |
| RetrieveUserByID | http://127.0.0.1:5000/user/4 | GET | Pass |
| RetrieveUserByEmail | http://127.0.0.1:5000/user/find_by_email?email=email2@example.com | GET | Pass |
| GetAllUsers | http://127.0.0.1:5000/user/all | GET | Pass |
| GetUserInformation | http://127.0.0.1:5000/user/info?user_id=2 | GET | Pass |
| RetrieveUsersbyRole | http://127.0.0.1:5000/user/find_by_role?role=5 | GET | Pass |
| retrieve_password/ | http://127.0.0.1:5000/user/find_by_email?email=admin | GET | Pass |
| UdateUser | http://127.0.0.1:5000/user/update/9 | PUT | Pass |
| Search User | http://127.0.0.1:5000/user/search?page=1&per_page=10 | GET | Pass |
| ValidateUser | http://127.0.0.1:5000/user/validate | POST | Pass |
| UserTokenExpiration | http://127.0.0.1:5000/user/check_token | GET | Pass |
| GetAllFarmers | http://127.0.0.1:5000/user/farmer | GET | Pass |
| GetFarmerDetailsById | http://127.0.0.1:5000/user/farmer/19 | GET | Pass |
| UpdateFarmer | http://127.0.0.1:5000/user/farmer/1 | PUT | Pass |
| DeleteFarmer | http://127.0.0.1:5000/user/farmer/100 | DELETE | Pass |
| SearchFarmers | http://127.0.0.1:5000/user/search_farmers?assigned_office_id=3 | GET | Pass |
| AddFarmer | http://127.0.0.1:5000/user/farmer | POST | Pass |
| GetFarmerDetailsAdvance | http://127.0.0.1:5000/user/farmer/details/9 | GET | Pass |
| AddFarm | http://127.0.0.1:5000/farm | POST | Pass |
| SearchFarms | http://127.0.0.1:5000/farm/search?page=1&per_page=10&type=crop&farm_id=1&address | GET | Pass |
| SearchCultivation | http://127.0.0.1:5000/cultivation/search?farm_id=1 | GET | Pass |
| AddCultivation | http://127.0.0.1:5000/cultivation/info | POST | Pass |
| SearchAidDistribution | http://127.0.0.1:5000/aid/aid-distribution/search | GET | Pass |

| | | | |
|---|---|---|---|
| AddAidDistribution | http://127.0.0.1:5000/aid/aid-distribution | POST | Pass |
| sendMail | http://127.0.0.1:5000/communication/send | GET | Pass |
| AddAddress | http://127.0.0.1:5000/communication/address | POST | Pass |
| GEtAllAddress | http://127.0.0.1:5000/communication/address | GET | Pass |
| SearchAddressById | http://127.0.0.1:5000/communication/address/search?user_id=100 | GET | Pass |
| DeleteAddressByAddressID | http://127.0.0.1:5000/communication/address/36 | DELETE | Pass |
| UpdateAddressByAddress_Id | http://127.0.0.1:5000/communication/address/update/2 | POST | Pass |
| AddContacts | http://127.0.0.1:5000/communication/contacts | POST | Pass |
| DeleteContactbyContactId | http://127.0.0.1:5000/communication/contacts/delete/2 | DELETE | Pass |
| SearchContact | http://127.0.0.1:5000/communication/contacts/search?user_id=2 | GET | Pass |

Some screen shots are included for evidence.

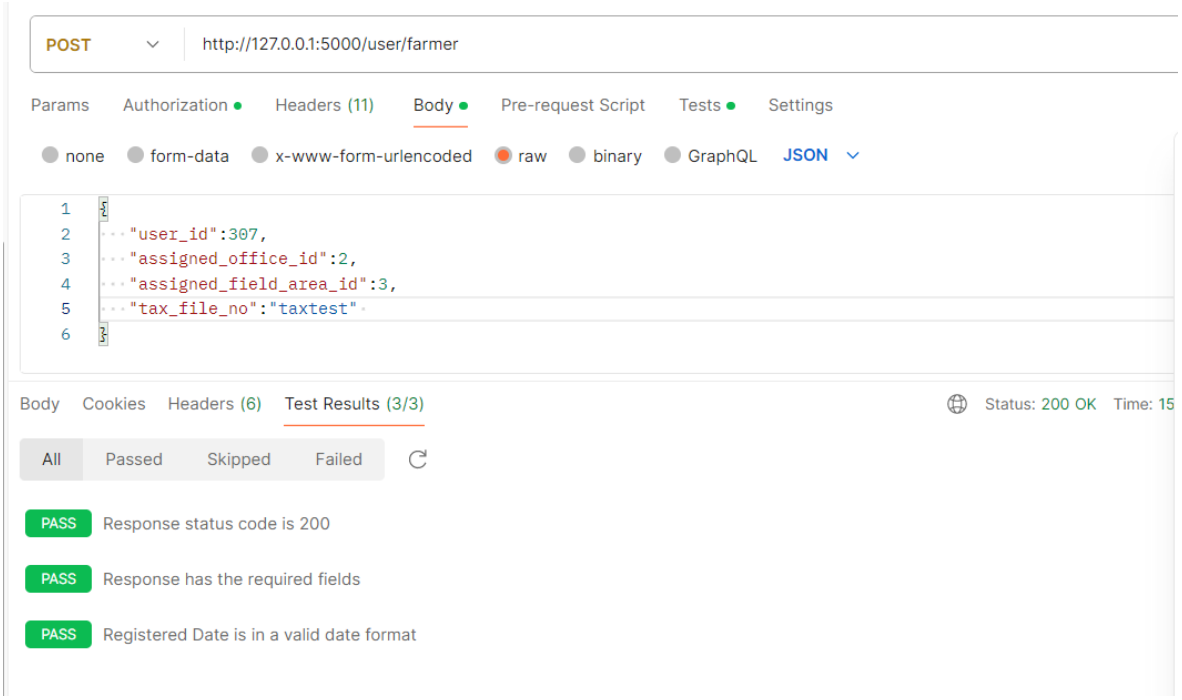



*Figure 1: Add farmer test*

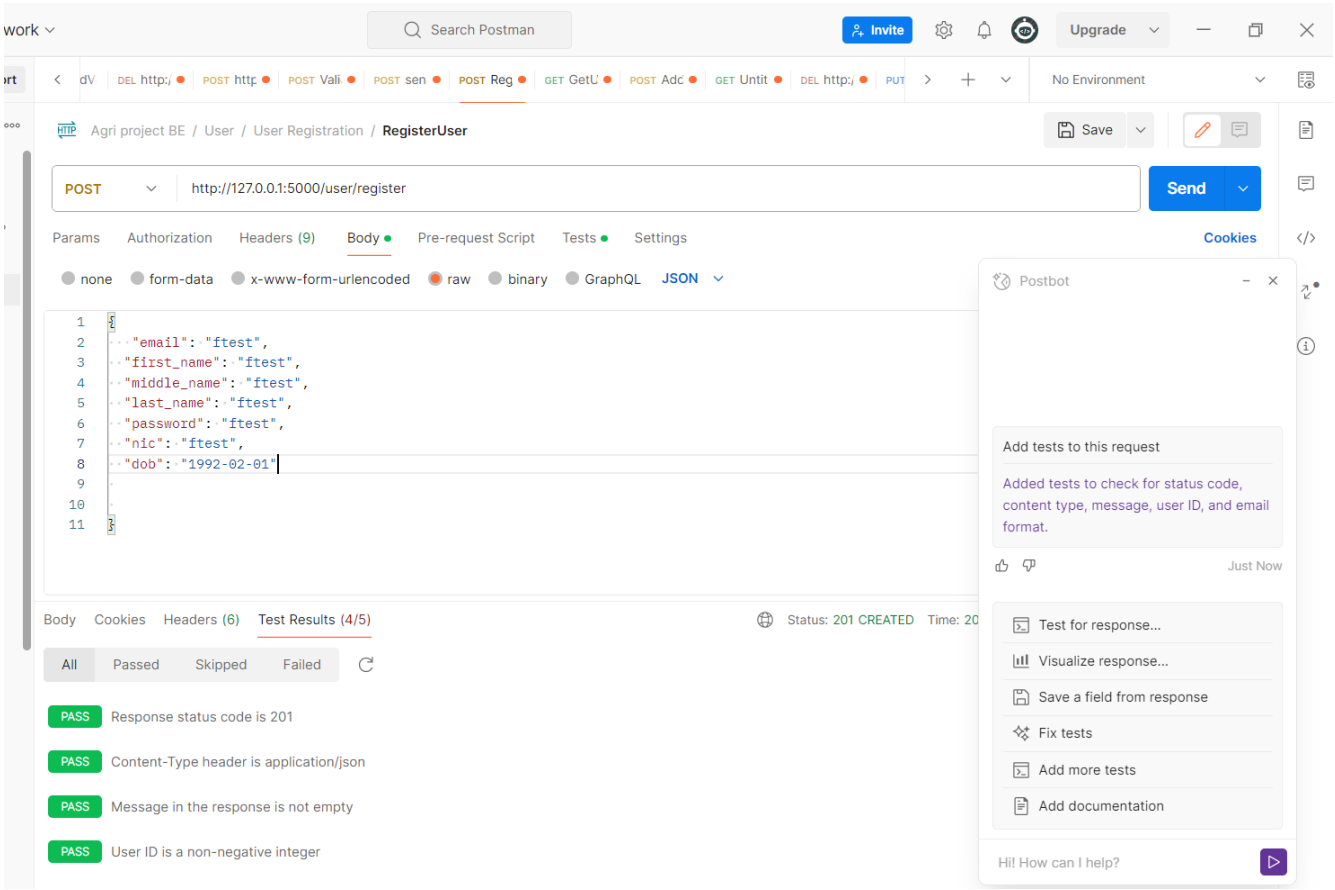



*Figure 3: User Registration Testing*

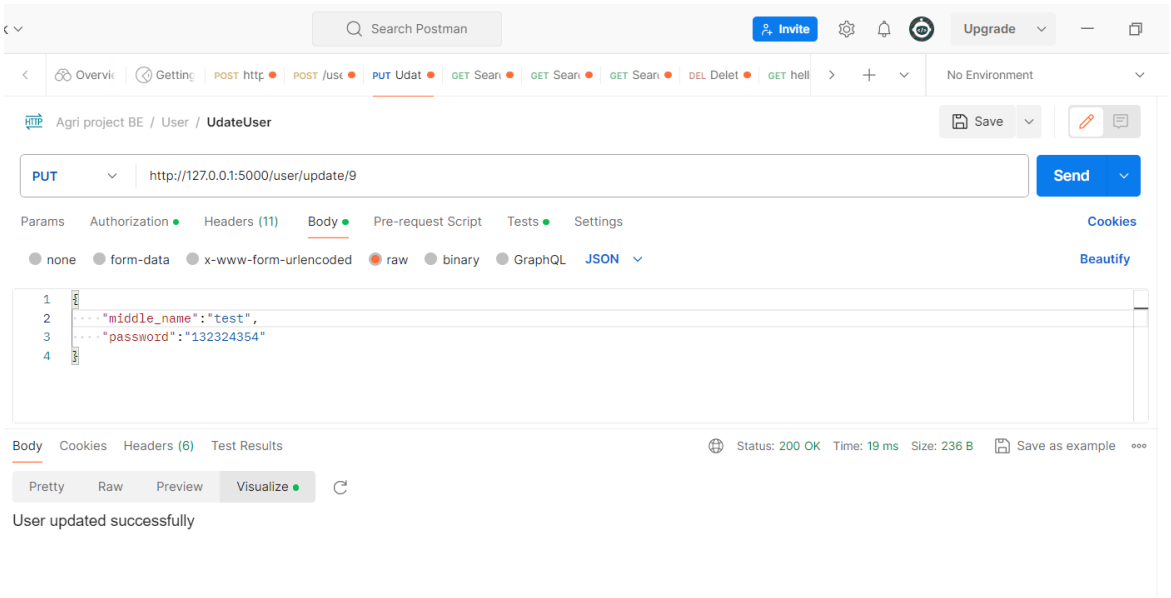



*Figure 2: Update user testing*

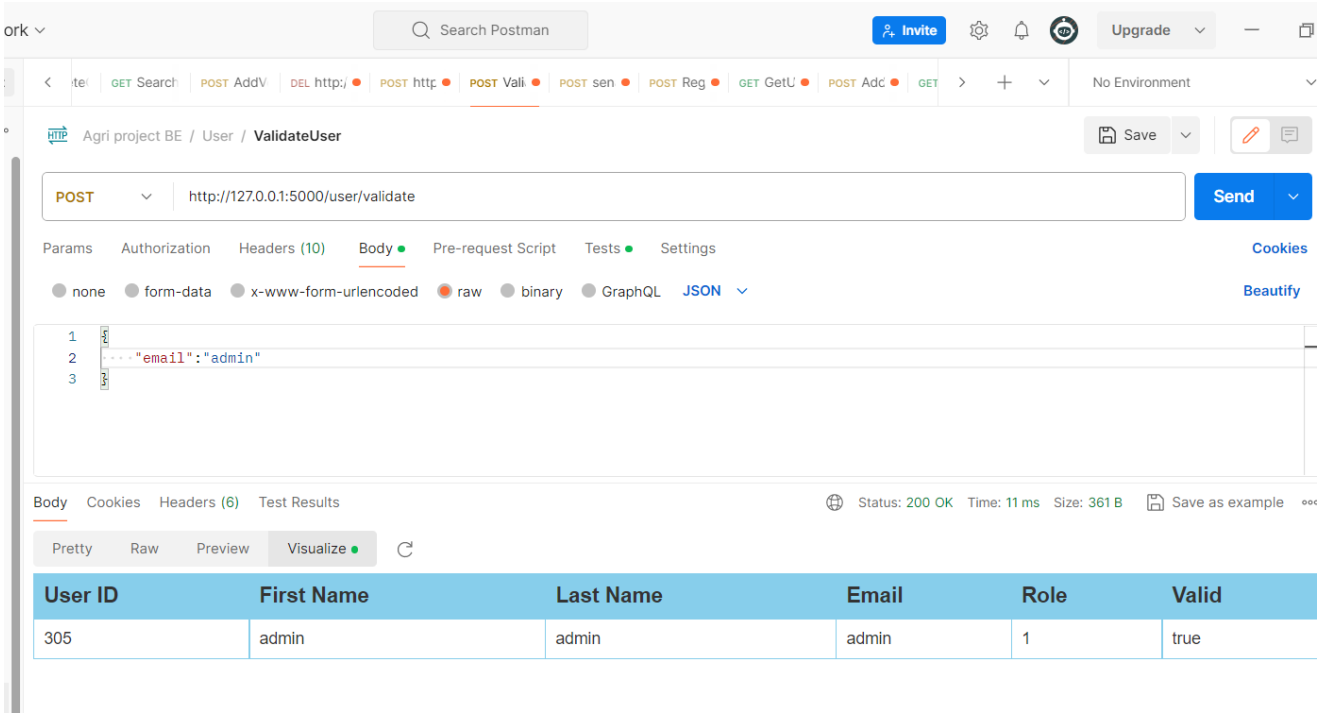



*Figure 4: User validation Test*

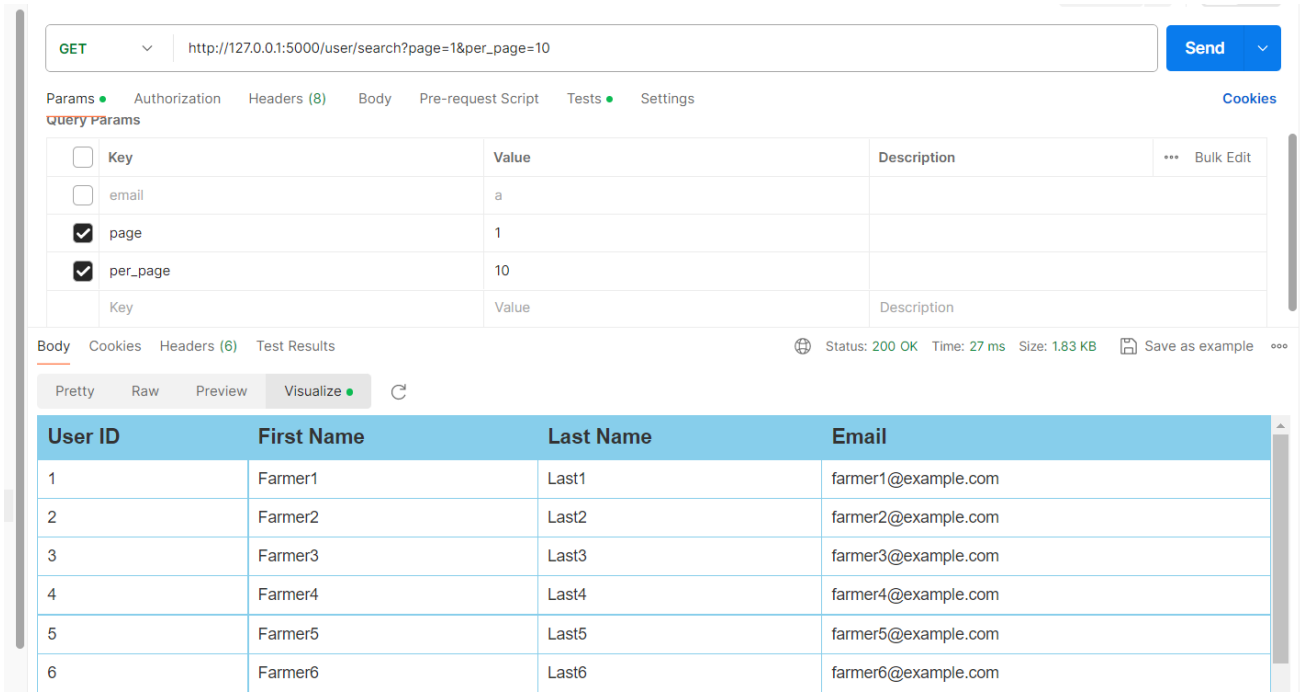



*Figure 5: Search User Testing*

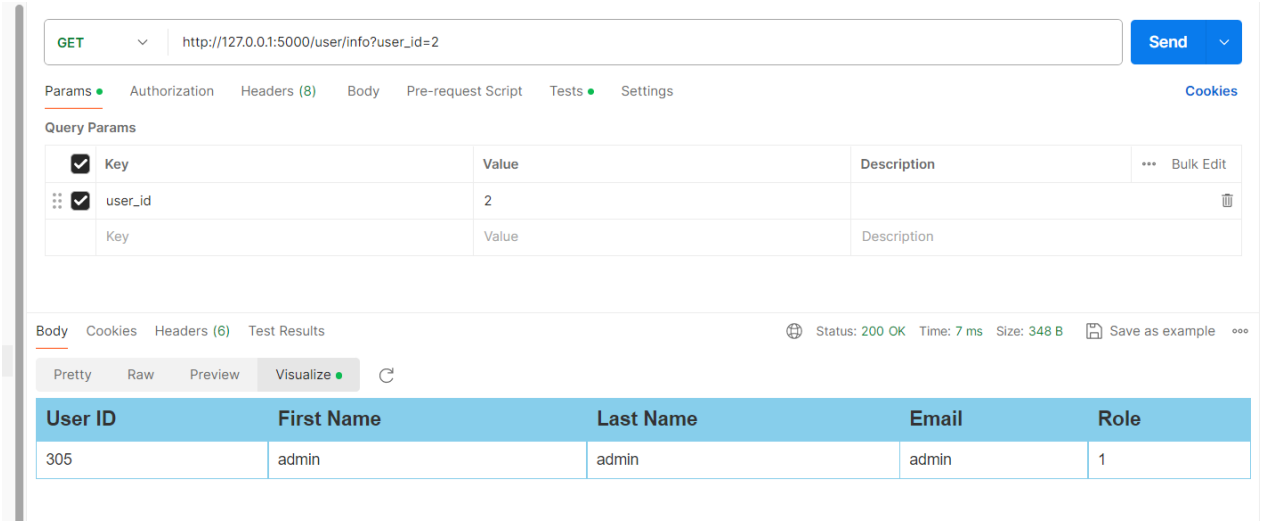



*Figure 6: Get User Info By user Id Test*

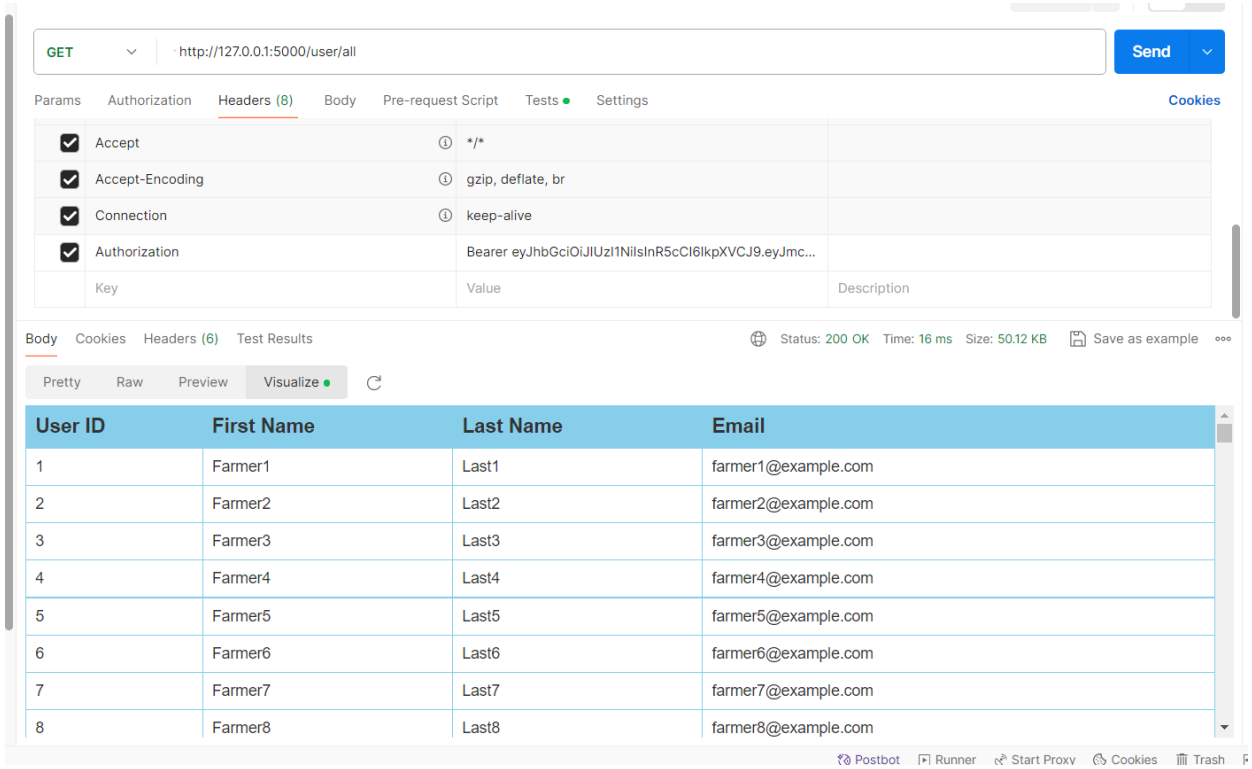



*Figure 8: Get All Users Test*

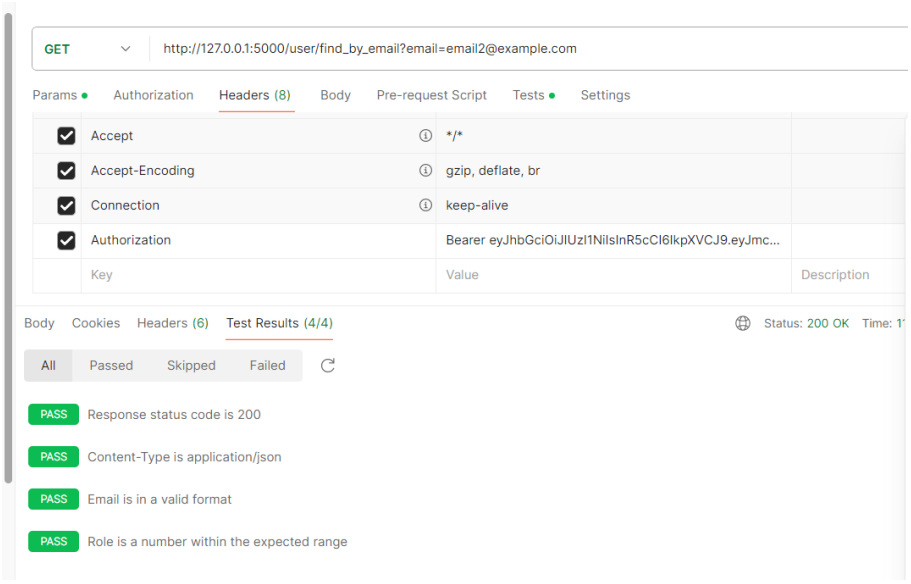



*Figure 7: Retrieve user by Email*

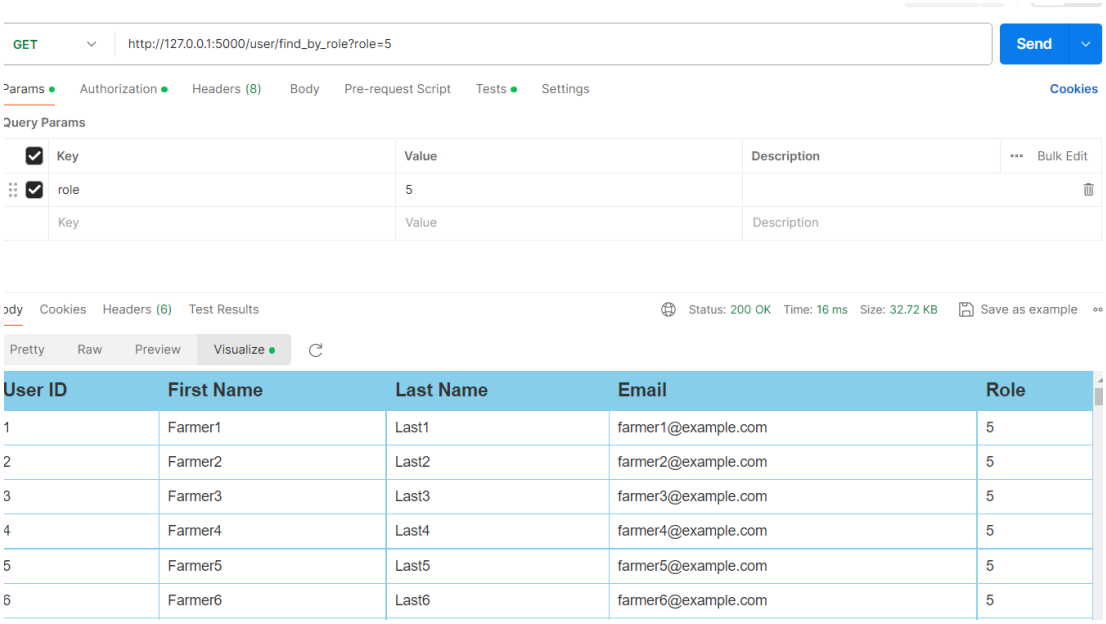



*Figure 9: Retrieve user by role Test*

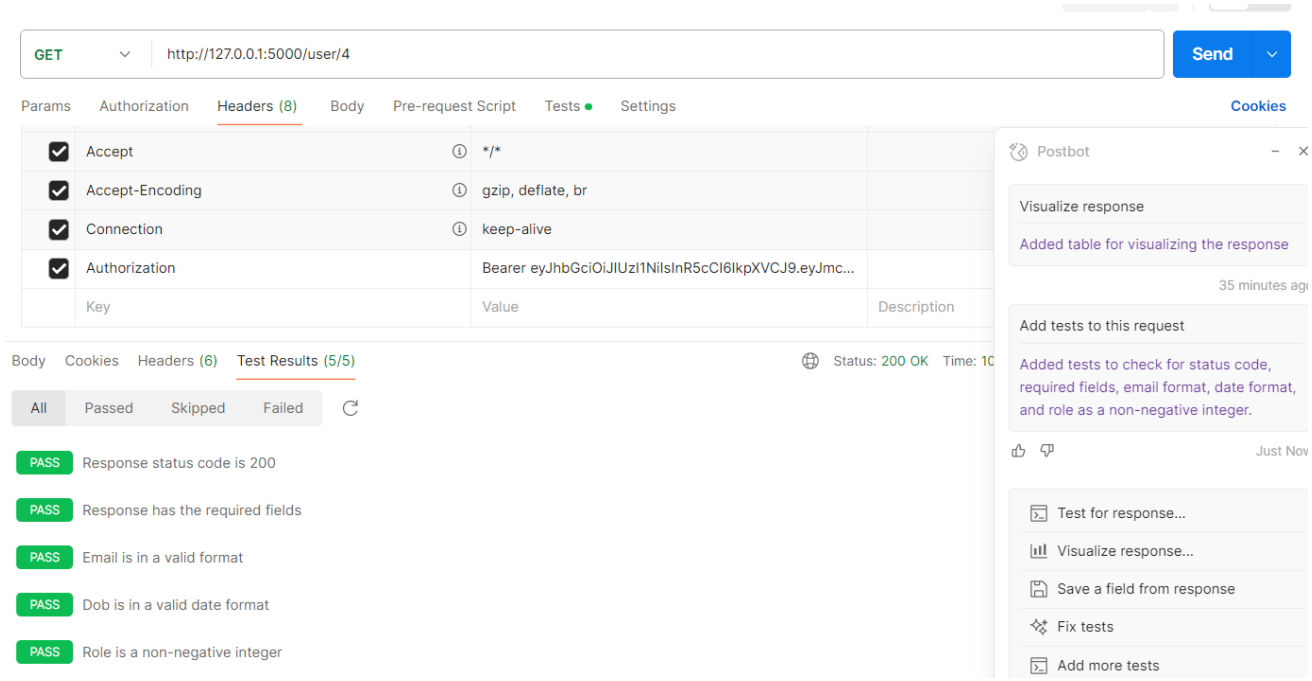



*Figure 10: Retrieve user by Id Test*

Appendix J

# "Ceylon AgriData" System Feedback Survey

***"Ceylon AgriData"*** is a cloud-based system aiding the government's agricultural sector by efficiently managing data for informed decision-making.
Please take a moment to share your feedback through this user feedback survey, providing honest insights on your experience with the prototype. Your contribution will greatly aid us in shaping a more impactful solution. **Thank you for your participation!**

* Indicates required question

1. Email *

2. What is your role within the agricultural sector? *

*Mark only one oval.*

- ◯ Administrator *Skip to question 3*
- ◯ Agriculture Field Officer *Skip to question 28*
- ◯ Farmer *Skip to question 61*
- ◯ Researcher *Skip to question 75*
- ◯ Other *Skip to question 84*

## Administrator Ctd.

3. First Name *

4. Last Name *

5. Your District

_____

6. **Registration** *

How straightforward was user registration process on "Ceylon AgriData"?

*Mark only one oval per row.*

| | Extremely Challenging | Challenging | Neutral | Straightforward | Extremely Straightforward |
|---|---|---|---|---|---|
| **how straightforward was user registration process on "Ceylon AgriData"?** | ◯ | ◯ | ◯ | ◯ | ◯ |

7. Were there any **obstacles** encountered during the registration process? If yes, please share details. *

_____

_____

_____

_____

_____

"Ceylon AgriData" WebApp

8. How seamless was the process of **entering** agricultural data into "Ceylon AgriData"? *

*Mark only one oval per row.*

| | Poor | Fair | Average | Good | Excellent |
|---|---|---|---|---|---|
| **Entering agricultural data into "Ceylon AgriData"** | ◯ | ◯ | ◯ | ◯ | ◯ |

9. Were there any **obstacles** encountered during the entering data into system? If yes, please share details. *

10. How seamless was the process of **updating** agricultural data into "Ceylon AgriData"? *

*Mark only one oval per row.*

| | Poor | Fair | Average | Good | Excellent |
|---|---|---|---|---|---|
| **Updating agricultural data** | ◯ | ◯ | ◯ | ◯ | ◯ |

11. Were there any **obstacles** encountered during the updating data into system? If yes, please share details. *

12. How seamless was the process of **deleting** agricultural data into "Ceylon AgriData"? *

*Mark only one oval per row.*

| | Poor | Fair | Average | Good | Excellent |
|---|---|---|---|---|---|
| **Deleting agricultural data** | ◯ | ◯ | ◯ | ◯ | ◯ |

13. Were there any **obstacles** encountered during the deleting data into system? If yes, please share details. *

14. How seamless was the process of **searching** agricultural data into "Ceylon AgriData"? *

*Mark only one oval per row.*

| | Poor | Fair | Average | Good | Excellent |
|---|---|---|---|---|---|
| **Searching agricultural data** | ◯ | ◯ | ◯ | ◯ | ◯ |

15. Were there any **obstacles** encountered during the searching data into system? If yes, please share details. *

16. How seamless was the process of **User Management** "Ceylon AgriData"? *

*Mark only one oval per row.*

| | Poor | Fair | Average | Good | Excellent |
|---|---|---|---|---|---|
| **User Management** | | | | | |

17. Were there any **obstacles** encountered during the user management data into system? If yes, please share details. *

18. Effectiveness of Reporting *

*Mark only one oval per row.*

| | Ineffective | Limited | Adequate | Effective | Highly Effective |
|---|---|---|---|---|---|
| **Efficacy of Reports Feature** | | | | | |

19. Did you face any **difficulties** while setting parameters for reports? Please describe your experience *

20. Effectiveness of Free Advertising Feature *

*Mark only one oval per row.*

| | Ineffective | Poor | Satisfactory | Effective | Highly Effective |
|---|---|---|---|---|---|
| **Efficacy of Feature** | ◯ | ◯ | ◯ | ◯ | ◯ |

21. Did you face any **difficulties** while engaging in the above service? Please Describe your Experience. *

22. Effectiveness of Message Broadcasting Service *

*Mark only one oval per row.*

| | Ineffective | Limited | Adequate | Effective | Highly Effective |
|---|---|---|---|---|---|
| **Effectiveness of Service** | ◯ | ◯ | ◯ | ◯ | ◯ |

23. Did you face any **difficulties** while engaging in the above service? Please Describe your Experience. *

24. How would you rate the UI design of "Ceylon AgriData" ? *

*Mark only one oval per row.*

| | Poor | Fair | Average | Good | Excellent |
|---|---|---|---|---|---|
| **UI Design** | ◯ | ◯ | ◯ | ◯ | ◯ |

25. Suggestions for enhancing the UI design. * Your opinions are welcome!

26. How would you rate your **overall satisfaction** with Ceylon AgriData's user-friendliness? *

*Mark only one oval per row.*

| | Fairly Unsatisfied | Neutral | Satisfied |
|---|---|---|---|
| **Overall satisfaction** | ◯ | ◯ | ◯ |

27. If you have any suggestions, please specify.

_____

## Agriculture Field Officer Ctd.

28. First Name *

_____

29. Last Name *

_____

30. Your District

_____

31. **Registration** *

How straightforward was user registration process on "Ceylon AgriData"?

*Mark only one oval per row.*

| | Extremely Challenging | Challenging | Neutral | Straightforward | Extremely Straightforward |
|---|---|---|---|---|---|
| **how straightforward was user registration process on "Ceylon AgriData"?** | | | | | |

32. Were there any **obstacles** encountered during the registration process? If yes, please share details. *

"Ceylon AgriData" WebApp

33. How seamless was the process of **entering** agricultural data into "Ceylon AgriData"? *

*Mark only one oval per row.*

| | Poor | Fair | Average | Good | Excellent |
|---|---|---|---|---|---|
| **Entering agricultural data into "Ceylon AgriData"** | | | | | |

34. Were there any **obstacles** encountered during the entering data into system? If yes, please share details. *

35. How seamless was the process of **updating** agricultural data into "Ceylon AgriData"? *

*Mark only one oval per row.*

| | Poor | Fair | Average | Good | Excellent |
|---|---|---|---|---|---|
| **Updating agricultural data** | ◯ | ◯ | ◯ | ◯ | ◯ |

36. Were there any **obstacles** encountered during the updating data into system? If yes, please share details. *

37. How seamless was the process of **deleting** agricultural data into "Ceylon AgriData"? *

*Mark only one oval per row.*

| | Poor | Fair | Average | Good | Excellent |
|---|---|---|---|---|---|
| **Deleting agricultural data** | ◯ | ◯ | ◯ | ◯ | ◯ |

38. Were there any **obstacles** encountered during the deleting data into system? If yes, please share details. *

39. How seamless was the process of **searching** agricultural data into "Ceylon AgriData"? *

*Mark only one oval per row.*

| | Poor | Fair | Average | Good | Excellent |
|---|---|---|---|---|---|
| **Searching agricultural data** | ◯ | ◯ | ◯ | ◯ | ◯ |

40. Were there any **obstacles** encountered during the searching data into system? If yes, please share details. *

41. How seamless was the process of **User Management** "Ceylon AgriData"? *

*Mark only one oval per row.*

| | Poor | Fair | Average | Good | Excellent |
|---|---|---|---|---|---|
| **User Management** | ◯ | ◯ | ◯ | ◯ | ◯ |

42. Were there any **obstacles** encountered during the user management data into system? If yes, please share details. *

43. Effectiveness of Reporting *

*Mark only one oval per row.*

| | Ineffective | Limited | Adequate | Effective | Highly Effective |
|---|---|---|---|---|---|
| **Efficacy of Reports Feature** | | | | | |

44. Did you face any **difficulties** while setting parameters for reports? Please describe your experience *

45. Effectiveness of Free Advertising Feature *

*Mark only one oval per row.*

| | Ineffective | Poor | Satisfactory | Effective | Highly Effective |
|---|---|---|---|---|---|
| **Efficacy of Feature** | ◯ | ◯ | ◯ | ◯ | ◯ |

46. Did you face any **difficulties** while engaging in the above service? Please Describe your Experience. *

47. Effectiveness of Message Broadcasting Service *

*Mark only one oval per row.*

| | Ineffective | Limited | Adequate | Effective | Highly Effective |
|---|---|---|---|---|---|
| **Effectiveness of Service** | ◯ | ◯ | ◯ | ◯ | ◯ |

48. Did you face any **difficulties** while engaging in the above service? Please Describe your Experience. *

49. How would you rate the UI design of "Ceylon AgriData" WebApp? *

*Mark only one oval per row.*

| | Poor | Fair | Average | Good | Excellent |
|---|---|---|---|---|---|
| **UI Design** | ◯ | ◯ | ◯ | ◯ | ◯ |

50. Suggestions for enhancing the UI design.
Your opinions are welcome!

51. How would you rate your **overall satisfaction** with Ceylon AgriData's user-friendliness of WebApp? *

*Mark only one oval per row.*

| | Fairly Unsatisfied | Neutral | Satisfied |
|---|---|---|---|
| **Overall satisfaction** | ◯ | ◯ | ◯ |

"Ceylon AgriData" Mobile Application

52. "How would you rate the effectiveness of **data collection through the mobile application** of Agricloud?" *

*Mark only one oval per row.*

| | Ineffective | Somewhat Ineffective | Neutral | Effective | Highly Effective |
|---|---|---|---|---|---|
| **Data collection** | ◯ | ◯ | ◯ | ◯ | ◯ |

53. Did you face any **difficulties** while engaging in the mobile application? Please Describe your Experience. *

54. How would you rate the UI design of "Ceylon AgriData" mobile application?

*Mark only one oval per row.*

| | Poor | Fair | Average | Good | Excellent |
|---|---|---|---|---|---|
| **UI design of Mobile Application** | ◯ | ◯ | ◯ | ◯ | ◯ |

55. Suggestions for enhancing the UI design (Mobile Application). Your opinions are welcome!

56. How would you rate your experience with following features in "Ceylon AgriData" mobile application? *

*Mark only one oval per row.*

| | Very Dissatisfied | Dissatisfied | Neutral | Satisfied | Very Satisfied |
|---|---|---|---|---|---|
| **Precisely determine cultivation locations** | | | | | |
| **Broadcast messaging service** | | | | | |

57. Did you face any **difficulties**
while engaging in above features of the mobile application?
Please Describe your Experience.

58. How would you rate your **overall satisfaction** with Ceylon AgriData's user-friendliness of the mobile application? *

*Mark only one oval per row.*

| | Fairly Unsatisfied | Neutral | Satisfied |
|---|---|---|---|
| **Overall satisfaction** | | | |

59. "How would you rate your experience with registering in the 'Ceylon AgriData' system through the mobile application, if you did not have an account before logging in?" *

*Mark only one oval per row.*

| | Very Dissatisfied | Dissatisfied | Neutral | Satisfied | Very Satisfied |
|---|---|---|---|---|---|
| **Register through Mobile application** | | | | | |

60. If you have any further suggestions, Please specify.

## Farmer ctd.

61. First Name *

62. Last Name *

63. Your District

64. "How would you rate your experience with the Free Advertising Service feature on "Ceylon AgriData" WebApp? *

*Mark only one oval per row.*

| | Very Dissatisfied | Dissatisfied | Neutral | Satisfied | Very Satisfied |
|---|---|---|---|---|---|
| **Free Advertising Service** | ◯ | ◯ | ◯ | ◯ | ◯ |

65. Are you experiencing any difficulties with the Free Advertising Service?

*Mark only one oval.*

◯ Yes *Skip to question 66*

◯ No *Skip to question 68*

## Free Advertising Service

66. Could you please specify the difficulties you have faced with the Free Advertising Service? *

_____

_____

_____

_____

_____

67. If you have any suggestions, please specify

_____

## Farmer

68. How effective do you find the broadcast message service for receiving important updates such as fertilizer distribution and other essential messages *

*Mark only one oval per row.*

| | Not Effective | Somewhat Effective | Neutral | Effective | Very Effective |
|---|---|---|---|---|---|
| **Effectiveness of broadcast messaging service** | ◯ | ◯ | ◯ | ◯ | ◯ |

69. "How do you feel about creating an account when you want to publish an advertisement on "Ceylon AgriData" ? How effective and easy was the process for you? *

*Mark only one oval per row.*

| | Very Difficult | Difficult | Neutral | Easy | Very Easy |
|---|---|---|---|---|---|
| **Process is..** | ◯ | ◯ | ◯ | ◯ | ◯ |

70. Did you face any difficulties in the process?
Please ,specify.

71. "How effective do you find the content overview provided by the web app as a farmer?" *

*Mark only one oval per row.*

| | Not Effective | Somewhat Effective | Neutral | Effective | Very Effective |
|---|---|---|---|---|---|
| **Row 1** | ◯ | ◯ | ◯ | ◯ | ◯ |

72. How would you rate the UI design of "Ceylon AgriData" ? *

*Mark only one oval per row.*

| | Poor | Fair | Average | Good | Excellent |
|---|---|---|---|---|---|
| **UI Design** | ◯ | ◯ | ◯ | ◯ | ◯ |

73. Suggestions for enhancing the UI design .
Your opinions are welcome!

74. If you have any further suggestions, Please specify.

Researcher

75. First Name *

76. Last Name *

77. Your District

78. **Requesting Non Public Data** *
"What was your experience with using the 'Contact Us' feature in the web application as a researcher to request non-publicly available data from the 'Ceylon AgriData' system?"

*Mark only one oval per row.*

| | Very Difficult | Difficult | Neutral | Easy | Very Easy |
|---|---|---|---|---|---|
| **Sign Up as a Researcher** | | | | | |

79. Did you face any difficulties in the process?
Please ,specify.

80. **Content Overview of Web App** *

How effective do you find the content overview provided by the web app in 'Ceylon AgriData'?

*Mark only one oval per row.*

| | Not Effective | Somewhat Effective | Neutral | Effective | Very Effective |
|---|---|---|---|---|---|
| **Content overview of WebApp** | ◯ | ◯ | ◯ | ◯ | ◯ |

81. **UI Design** How would you rate the UI design of 'Ceylon AgriData'?"

*Mark only one oval per row.*

| | Poor | Fair | Neutral | Good | Excellent |
|---|---|---|---|---|---|
| **UI Design** | ◯ | ◯ | ◯ | ◯ | ◯ |

82. Suggestions for enhancing the UI design .
Your opinions are welcome!

83. If you have any further suggestions, Please specify.

## Generic User

84. First Name *

_____

85. Last Name *

_____

86. District

_____

87. **WebApp content Engagement** *
Has the content provided by the web app in 'Ceylon AgriData' offered you any valuable services or information?

*Mark only one oval.*

- ◯ Yes *Skip to question 88*
- ◯ No *Skip to question 93*

## Generic User

88. **Reasons for Engagement** What motivates you to engage with the web app of 'Ceylon AgriData'? Please select all that apply. *

*Check all that apply.*

- ☐ Access to agricultural data
- ☐ News & updates
- ☐ Free Advertising service engagement
- ☐ Other: _____

89. **Requesting non public data** *

"What was your experience with using the 'Contact Us' feature in the web application as a researcher to request non-publicly available data from the 'Ceylon AgriData' system?"

*Mark only one oval per row.*

| | Very Difficult | Difficult | Neutral | Easy | Very Easy |
|---|---|---|---|---|---|
| **Sign Up as a Researcher** | ◯ | ◯ | ◯ | ◯ | ◯ |

90. Did you face any difficulties in the process?
Please ,specify.

_____

_____

_____

_____

_____

91. How do you find advertisements helpful to you on 'Ceylon AgriData'? Rate it. *

*Mark only one oval per row.*

| | Not helpful at all | Slightly helpful | Moderately helpful | Very helpful | Extremely helpful |
|---|---|---|---|---|---|
| **How advertisements helpful?** | ◯ | ◯ | ◯ | ◯ | ◯ |

92. If you have suggestions, please specify.

## Generic User

93. **UI Design and Suggestions** *

How would you rate the UI design of 'Ceylon AgriData'?

*Mark only one oval per row.*

| | Poor | Fair | Neural | Good | Execellent |
|---|---|---|---|---|---|
| **UI Design** | ◯ | ◯ | ◯ | ◯ | ◯ |

94. Suggestions for enhancing the UI design .
Your opinions are welcome!

95. If you have any suggestions for the improvement of the web app in 'Ceylon AgriData', please specify.

# Appendix K

## User Manual for "Ceylon AgriData" Mobile Application

*Ceylon AgriData* is a mobile application designed for Agriculture Field Officers to efficiently collect, manage, and report agriculture-related data of registered farmers, their farms, cultivation activities, issues reporting, and aid distribution. This user manual provides detailed instructions on how to use the *Ceylon AgriData* mobile application effectively.

## Content

## Technical Requirements:

- Mobile Platform: Android
- Android Version: Android 5.0 (and up) its dessert-themed code name: “Lollipop”.

## 1.Getting Started:

- Before using the "Ceylon AgriData" app, ensure that you have downloaded and installed it on your mobile device.
- Once installed, launch the app by tapping on its icon. You will be directed to a splashing page as Figure 1.

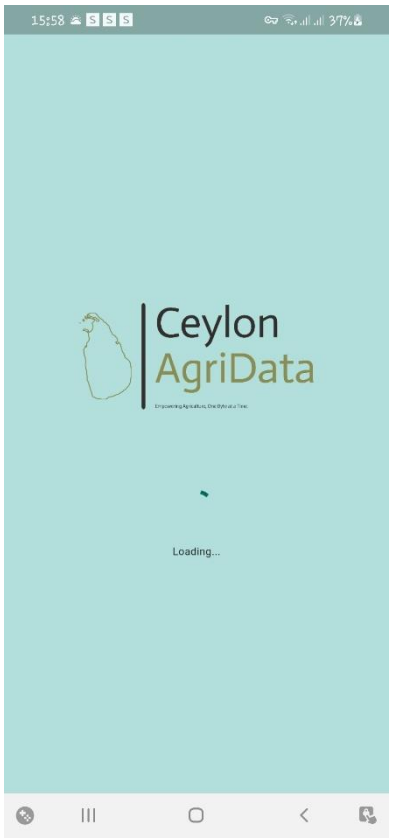



*Figure 1: Splashing Page*

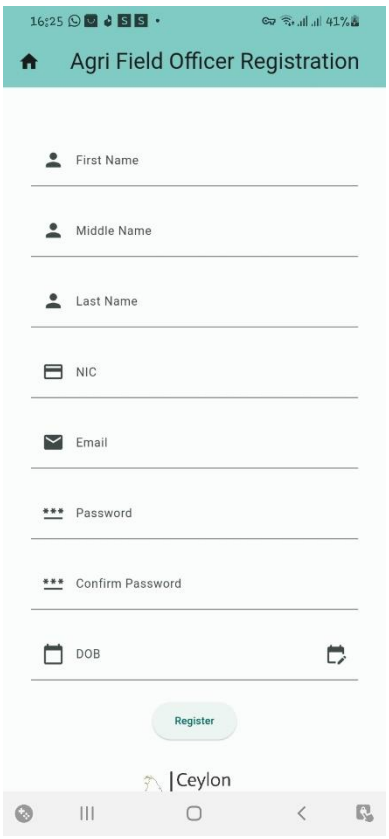



*Figure 2: Agri Field Officer Registration Page*

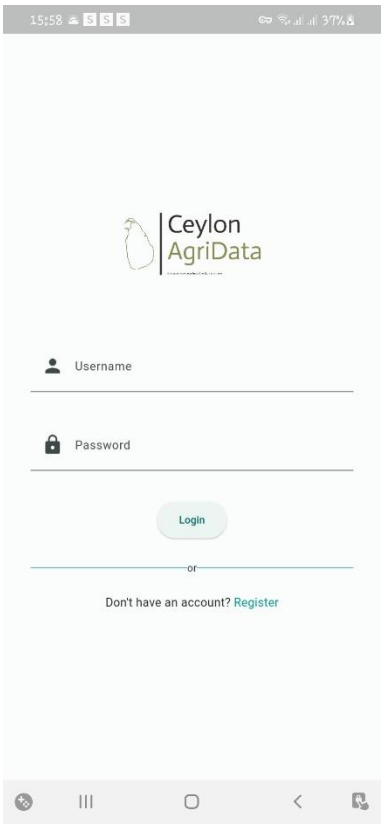



*Figure 3; User Login Page*

## 2. User Registration and Login

- If you are a new user, you will need to register by providing your username, password, and other required information as in Figure 2.
- After registration, your account will be pending approval from the admin.
- If you are already registered, simply log in using your username and password as Figure 3.

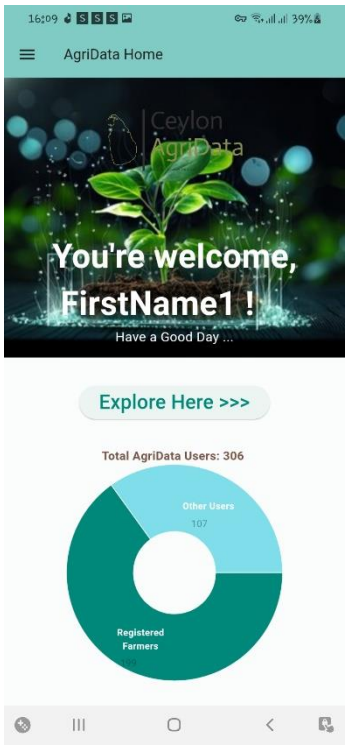



*Figure 4: Home Page*

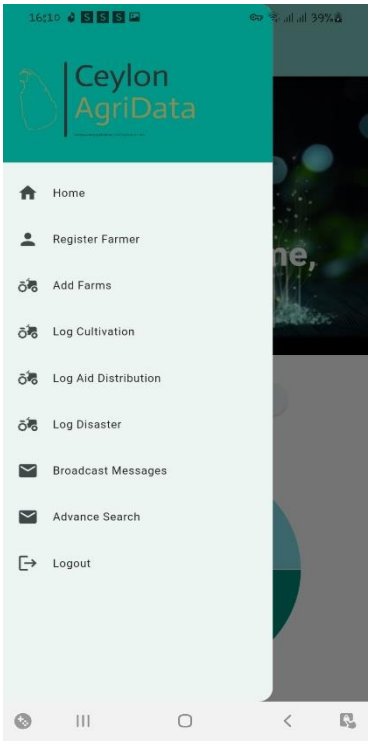



*Figure 5: Main Menu*


*Figure 6: Dash Board*

## 3. Home Screen

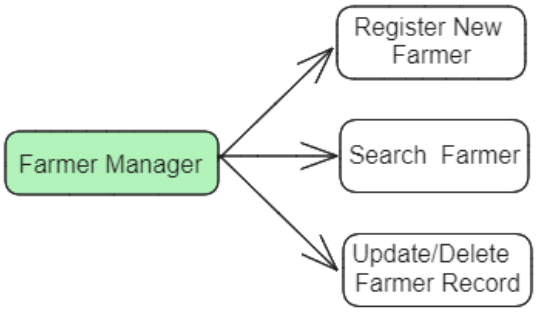
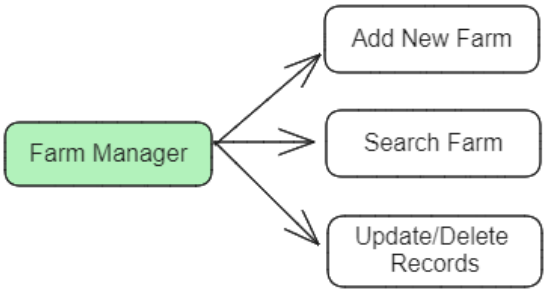
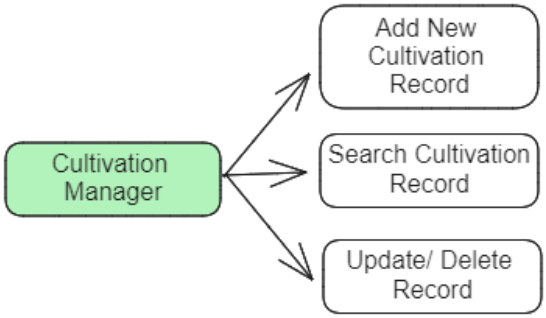
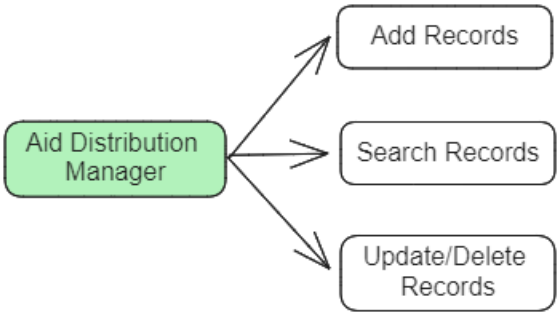
- Upon successful login, you will be directed to the home screen of the app
- From the home screen, you can access various features and functionalities of the app clicking “Explore More” button as in Figure 4.
- Or, you can use “Main Menu” for navigation to pages in Figure 5.

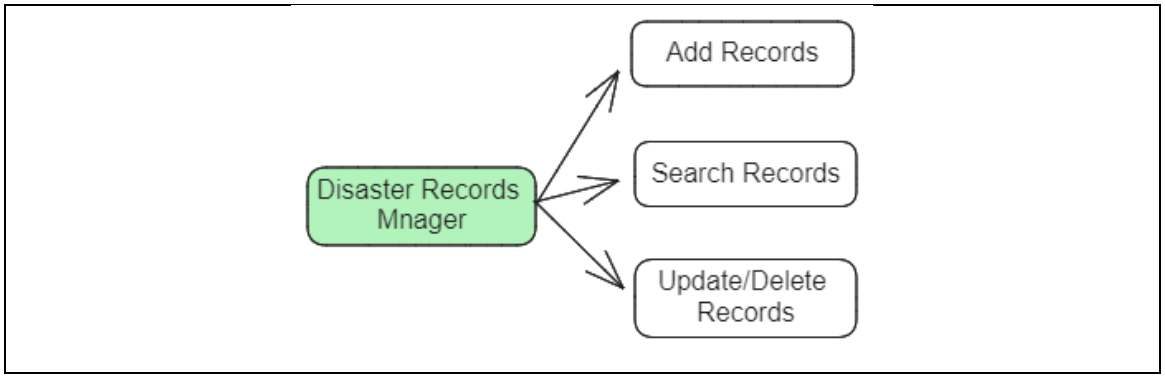
## 4.Dash Boad

- You can see the buttons to enroll the functionalities that this application is included, as in figure 6.

## 5. Collecting Agriculture Data

- To collect agriculture data, navigate to the respective section within the app as described in following table 1.
- Enter the required information such as farmer details, farm details, cultivation details, disaster records, aid distribution records etc.
- You can search, update, or delete records as needed.

| Dash Board Buttons | Buttons in Relevant pages to enroll particular functionalities |
|---|---|
|  | |
|  | |
|  | |
|  | |




## Farmer Registration

Fill the required information and click “Next”, then fill the section 2. And , click “Submit Button”. Figure 7 showcases the form. When you click sublit button, if the registration is success, you will be navigation to “Add Farm” section automatically. Or, you can use main menu at home page to navigate to “Register Farmer” section.

## Add Fam

Fill the information and click “Submit” button. Figure 8 indicates the particular form. When successfully added the record, you will be navigating to “Add cultivation information” Page. Or, you can navigate it right from the main menu at Home Page.

Use “Search User Id”, search button for your task gets completed.

## Add Cultivation Details

Fill the form in Figure 9 and press “Submit button”. Use “Get Current Location” button to get cultivation location. Refer Figure 10 for more information.

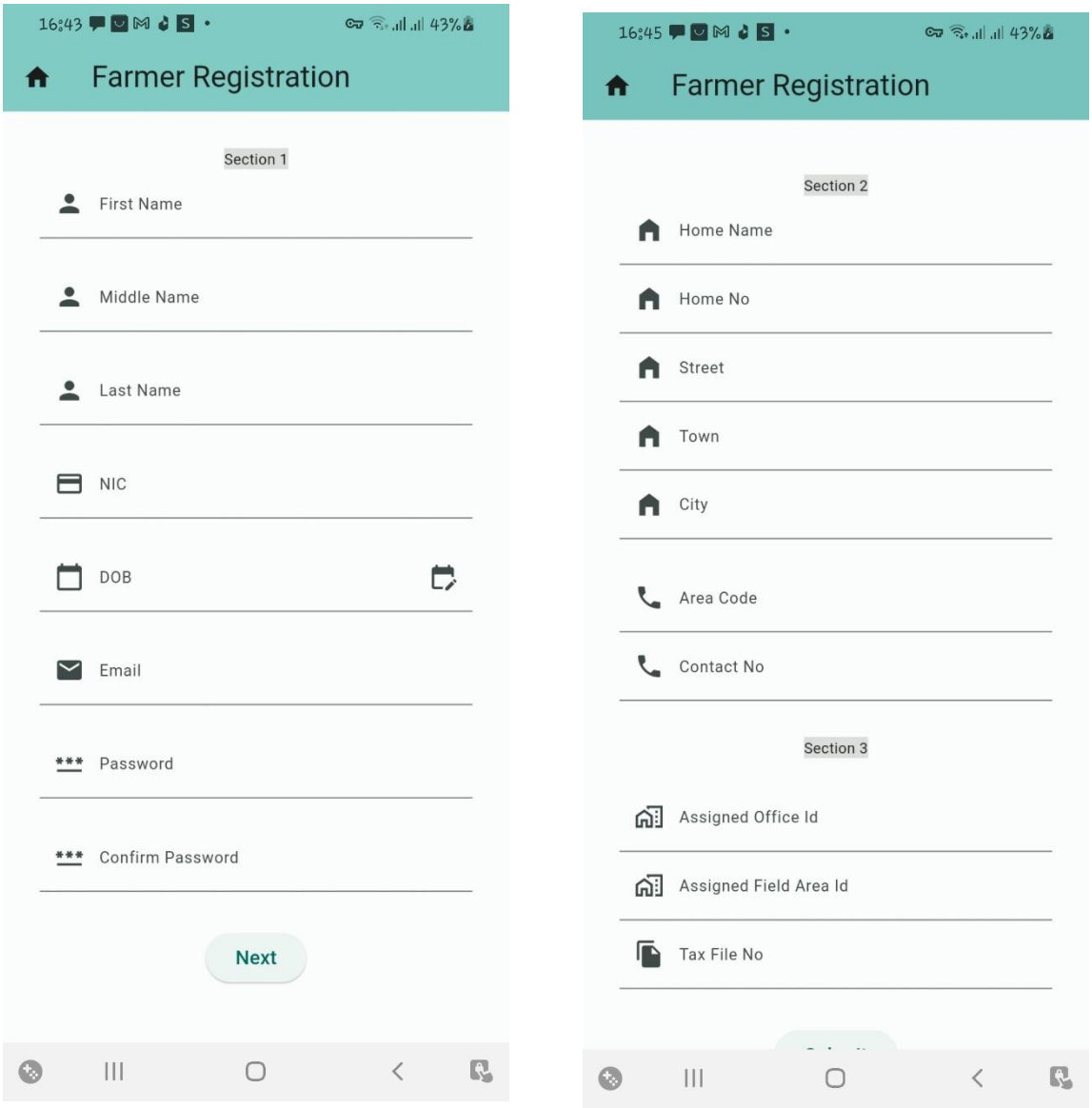



*Figure 7: Farmer Registration*

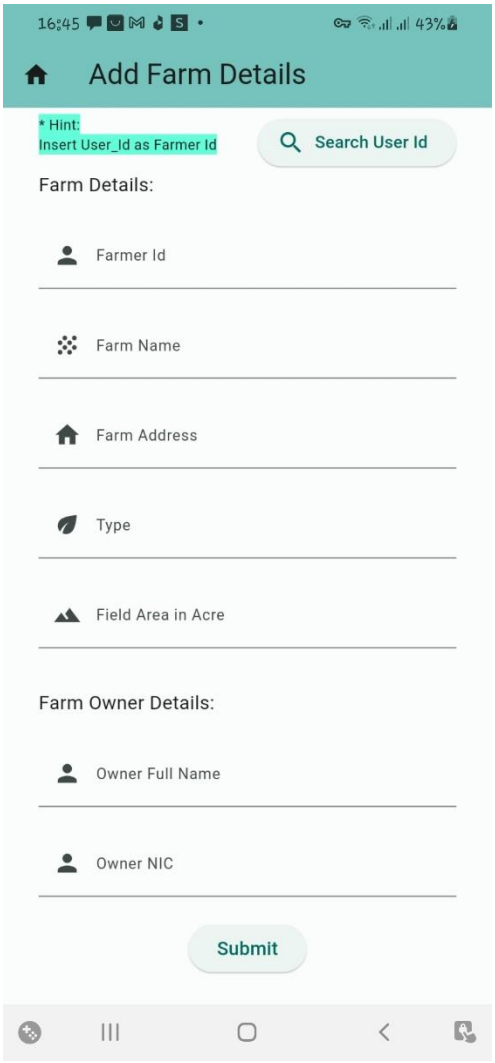



*Figure 8: Add Farms Page*

You can Choose relevant agriYear, Quartile etc. by the drop down.

Also, "Get Current Location" button will assist to get relevant location of the cultivation.

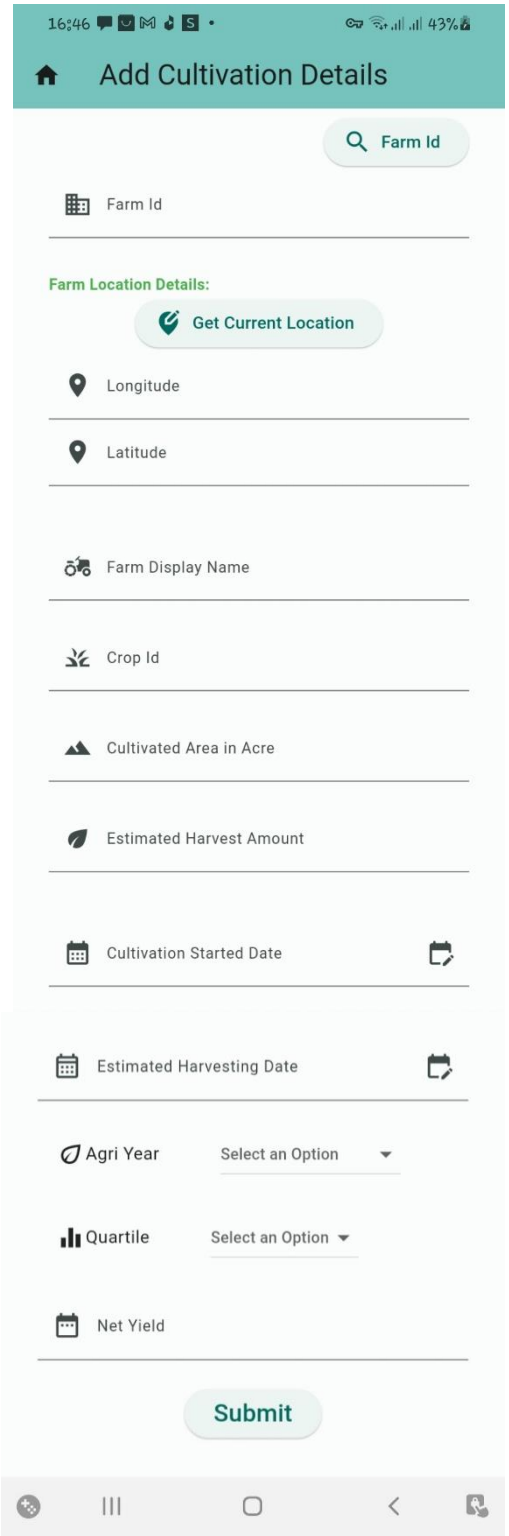



*Figure 9: Add Cultivation Information Page*

Right after click “Confirm Location” button, you will be returned to “Add Cultivation Details” page.

***Important: You should grant access to location fetching while using your mobile phone.

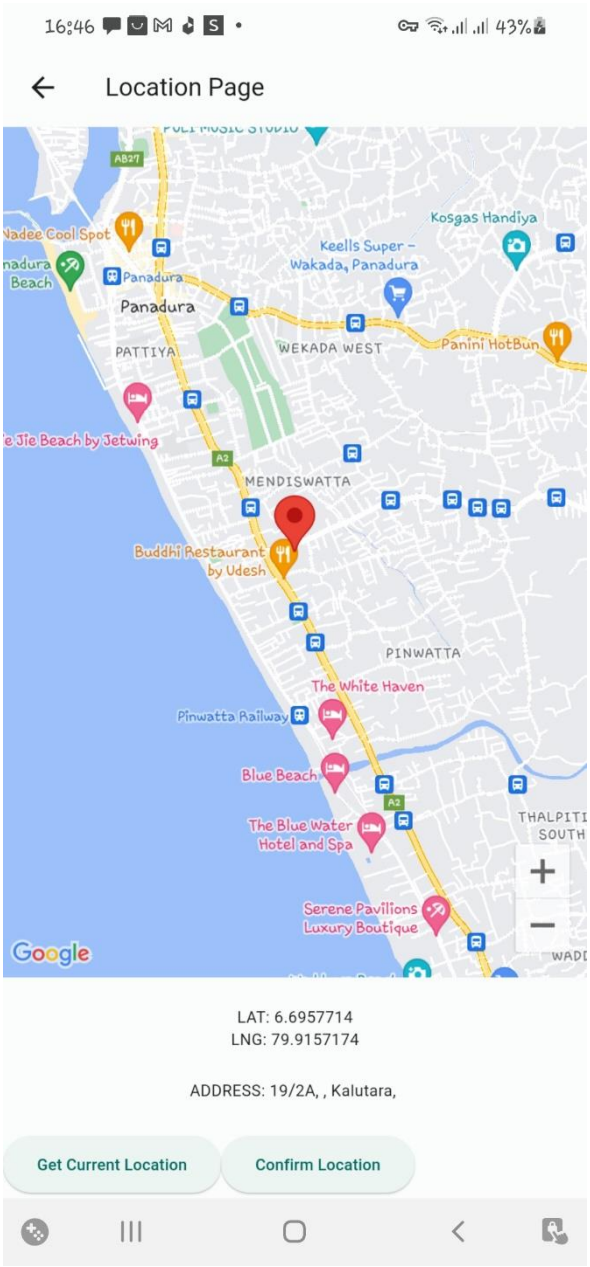




*Figure 10: Fetching current location*

## Add Aid Distribution Records

Use “Aid Id” as per your official documents. Fill the form in Figure 11 and click “Submit” button. Use the drop-down options to select “Aid Type”.

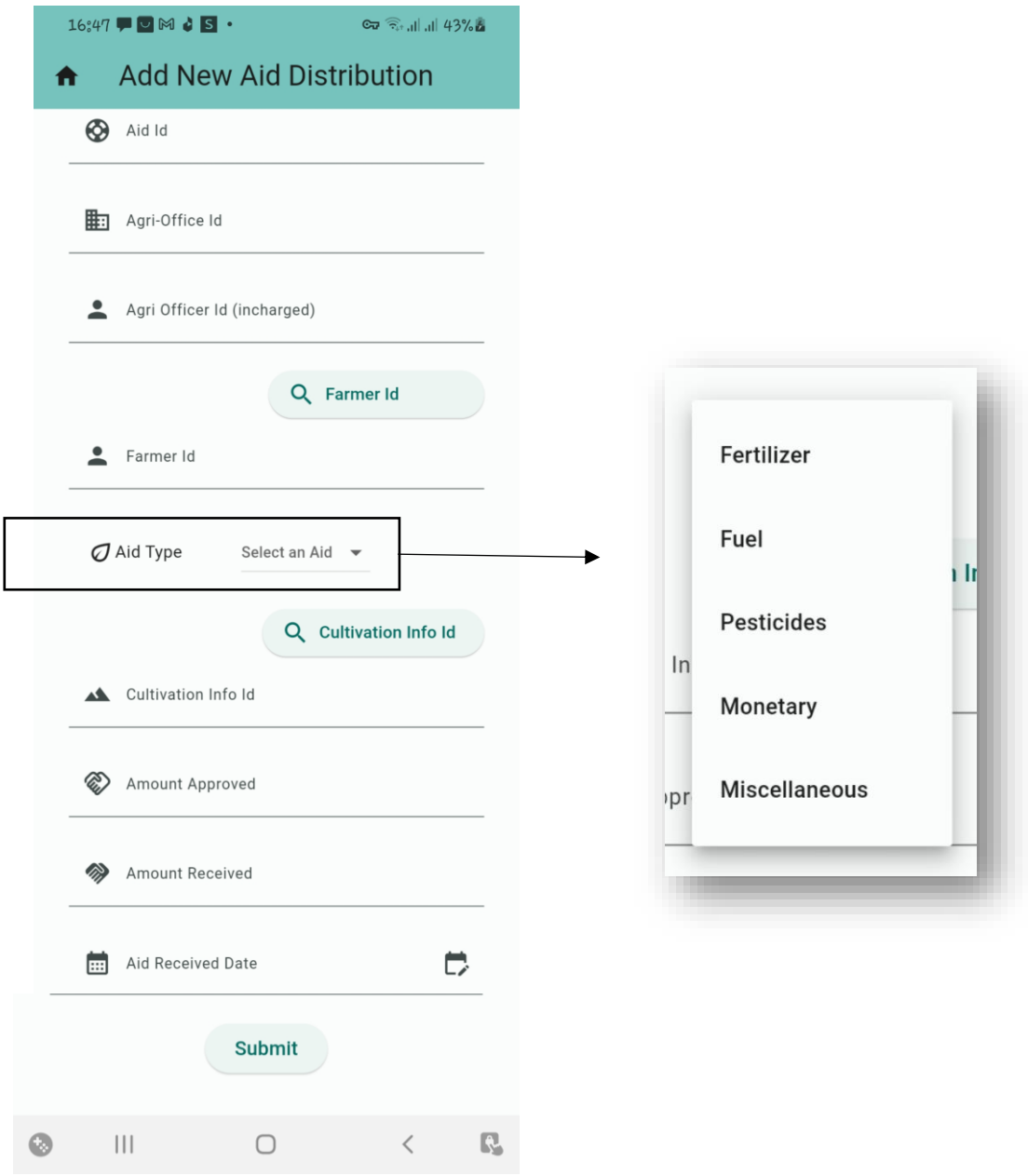



*Figure 11: Aid Distribution Page*

## Record Disaster Information

To complete the recording process, fill the form provided in Figure 12, and click the "Submit" button to proceed. Utilize the search button if you need to find specific information. Additionally, you may utilize the dropdown menus to select the type of disaster. You can navigate to this particular page form main menu at home page.

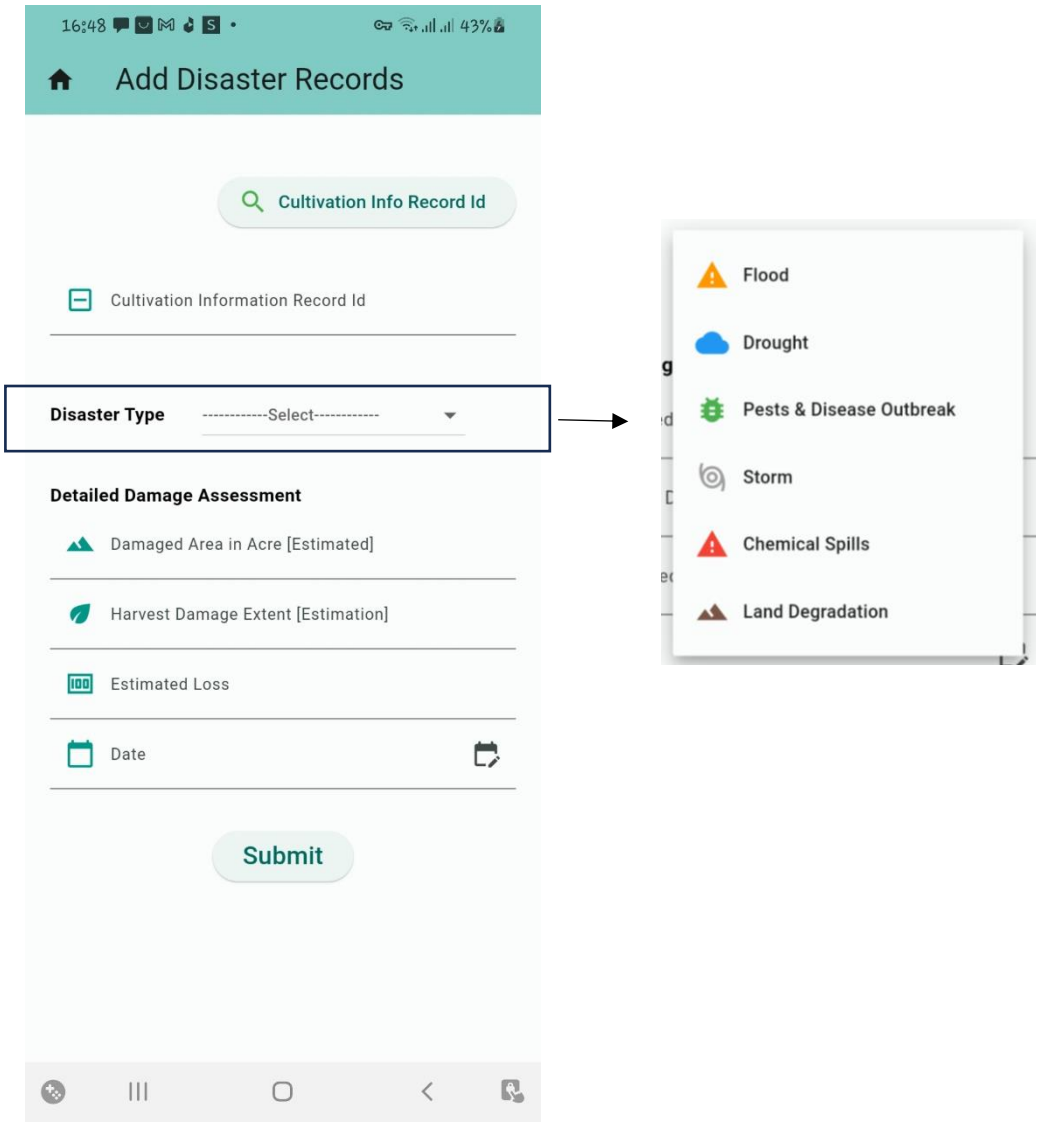



*Figure 12: Add Disaster Records Page*

## 6. Broadcast Message Service

With this feature, you have the ability to broadcast messages to registered farmers within the accredited area of agriculture field officers. To initiate a broadcast, craft a message with a suitable subject and body, then proceed by clicking the "Send" button. For reference, you can view the page layout in Figure 13. Additionally, there is an option to fetch the relevant email addresses of farmers by clicking the "All Farmer Emails" button.

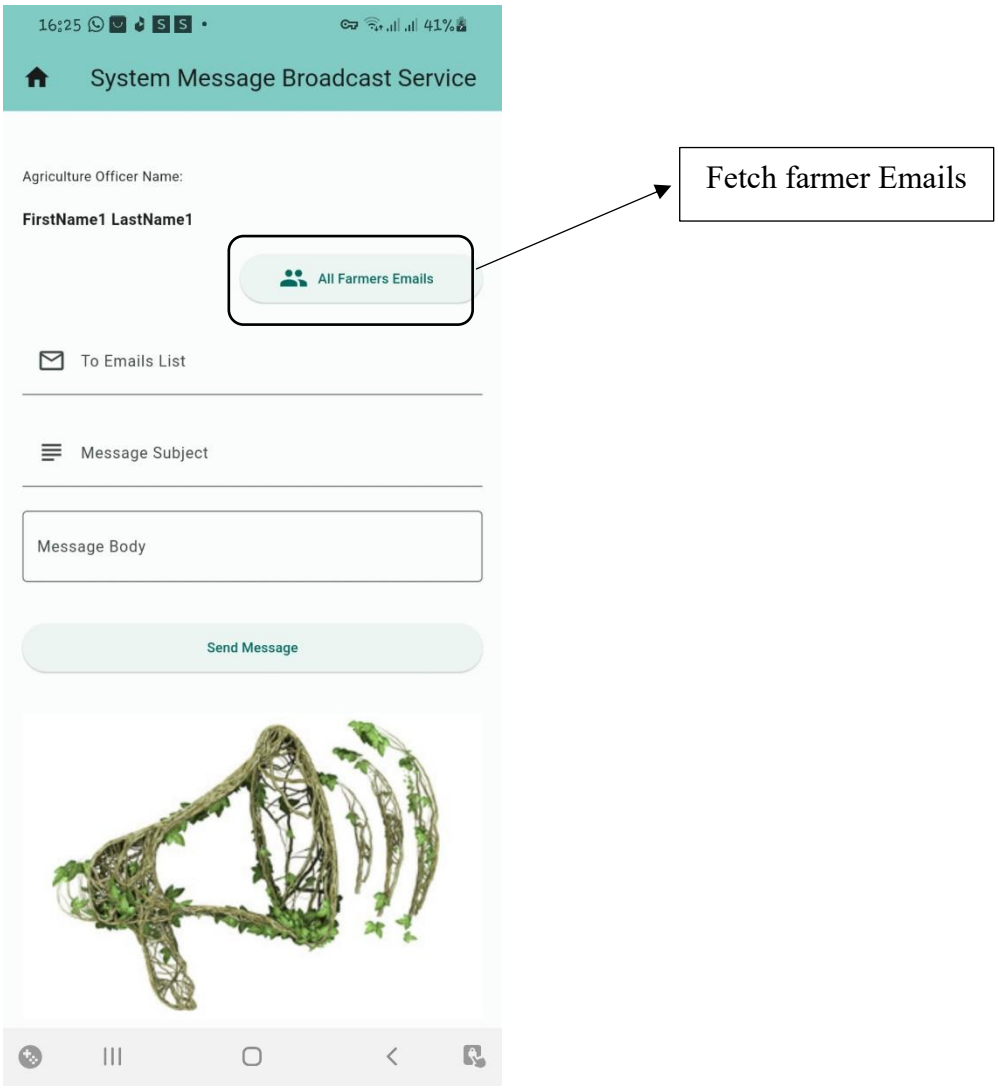



*Figure 13: Broadcast messages Page*

## 7. Manage Records

You have the ability to manage all inserted records by utilizing the SEARCH, UPDATE, and DELETE options provided. Figure 14 displays a group of pages dedicated to managing all

agriculture data discussed in Section 5. With these options, you can search for specific records, update existing information, or delete records as necessary.

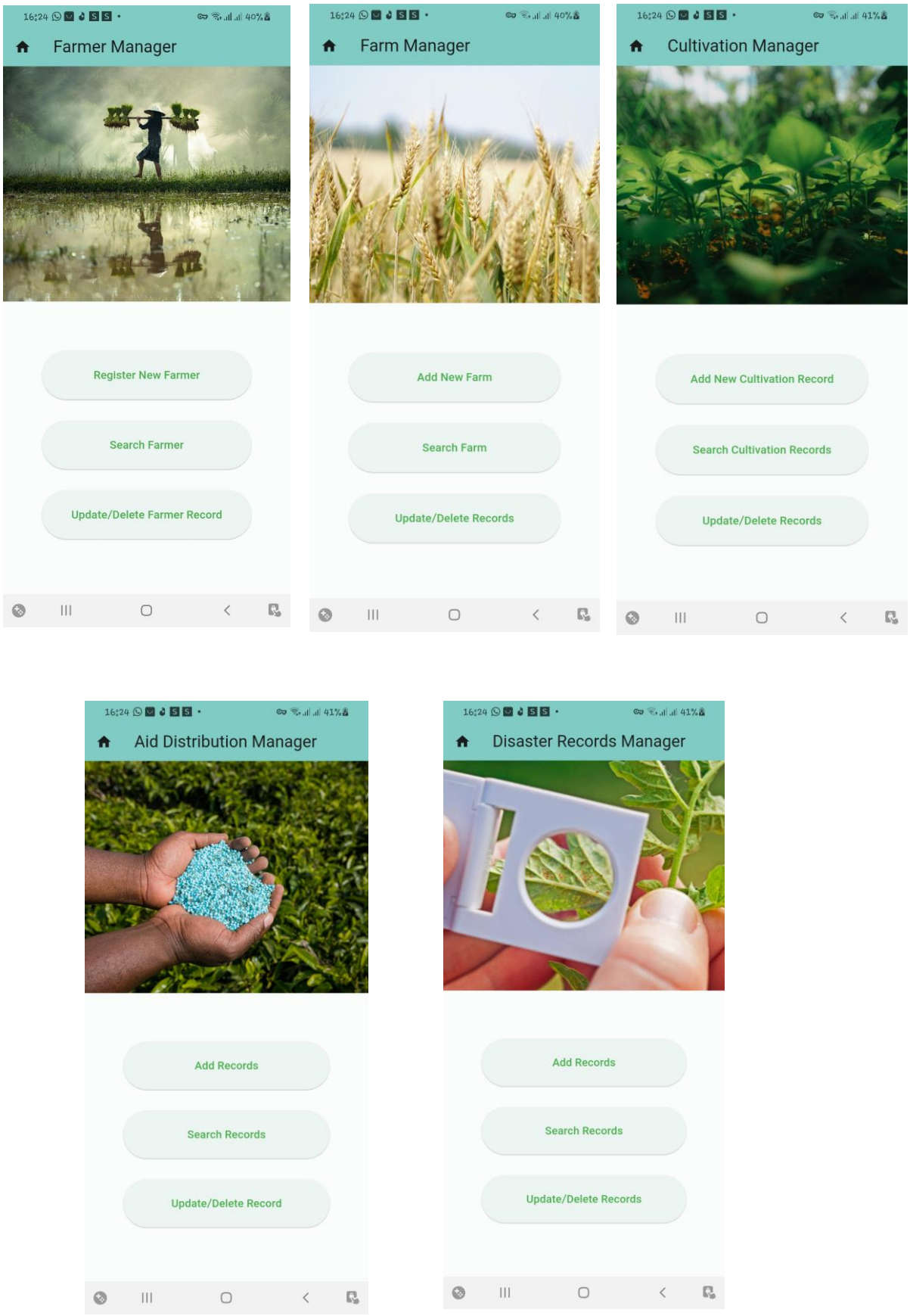



*Figure 14: Manage Records Pages*

Table 2 showcases the distinct pages for search functionality.

Use “Search” buttons to process search. Use “Clear Result” button to clear the fields. For example, see Figure 15. [available options: Search farmers, search farms, search aids, Search cultivation information]

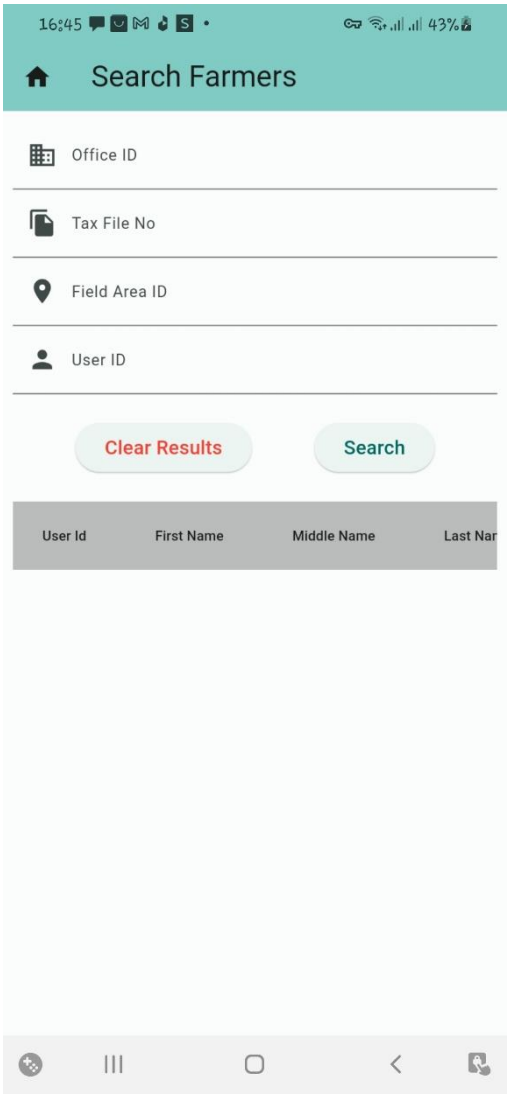



*Figure 15: Search Farmers*

## 8. Update/Delete Records

o update or delete records, simply click the "Update/Delete Records" button. Afterward, search for the desired records and preview them. You can then proceed to update the records accordingly. Alternatively, you can delete a record by clicking on the "delete" icon.

Figure 16 shows in detailed example. Enter a valid *Farmer_id* and tap the search icon

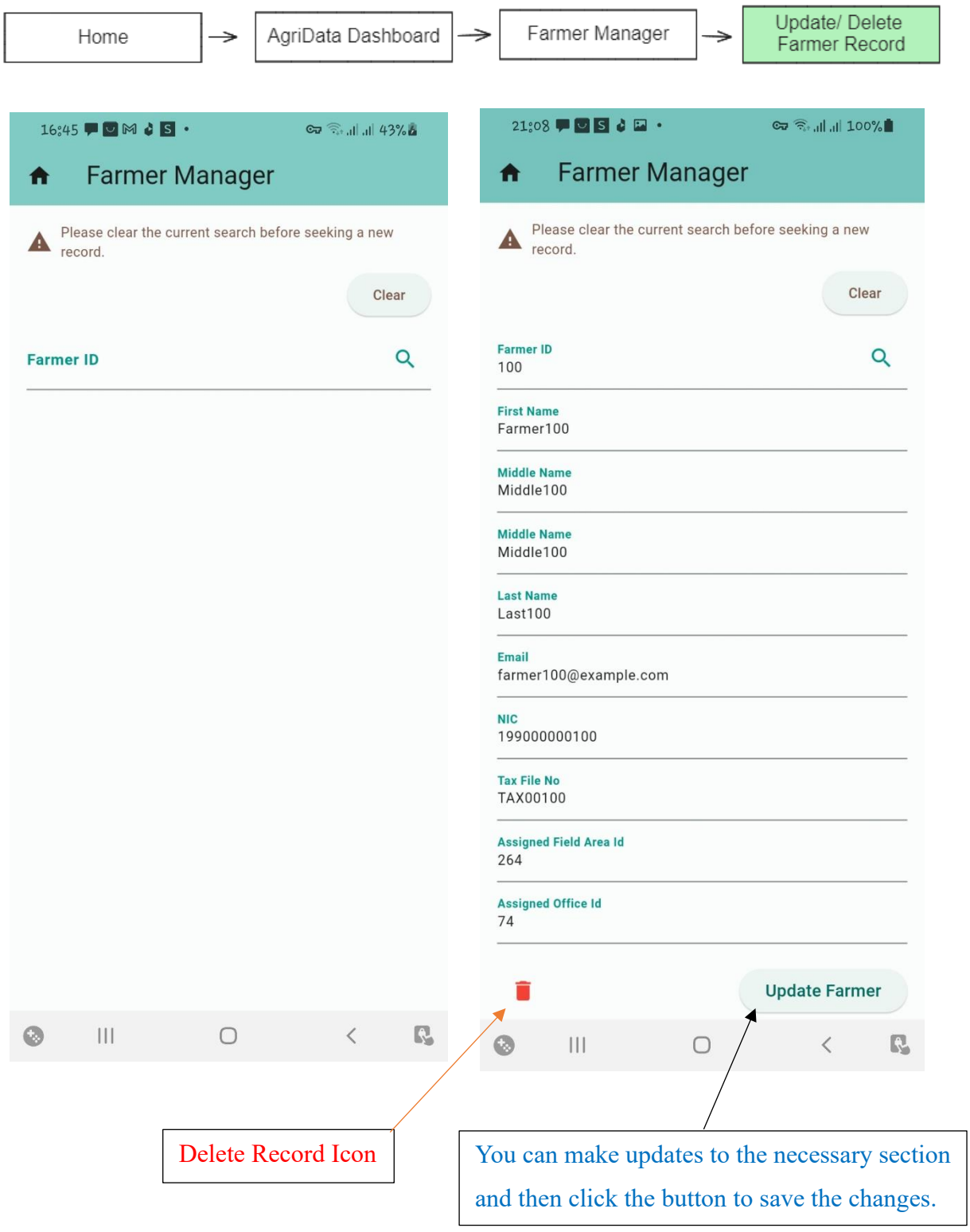



*Figure 16:Update / Delete Farmer Records*

## Additional Tips

- Ensure a stable internet connection for smooth operation of the app, especially when submitting data or broadcasting messages.
- Familiarize yourself with the app's interface and features to maximize efficiency during field visits.
- Contact the admin or technical support team for any assistance or issues related to the app.
- With the "Ceylon AgriData" mobile application, you can effectively manage agriculture data, report issues, distribute aids, and communicate with farmers, all from the convenience of your mobile device.

**HAPPY FARMING!**

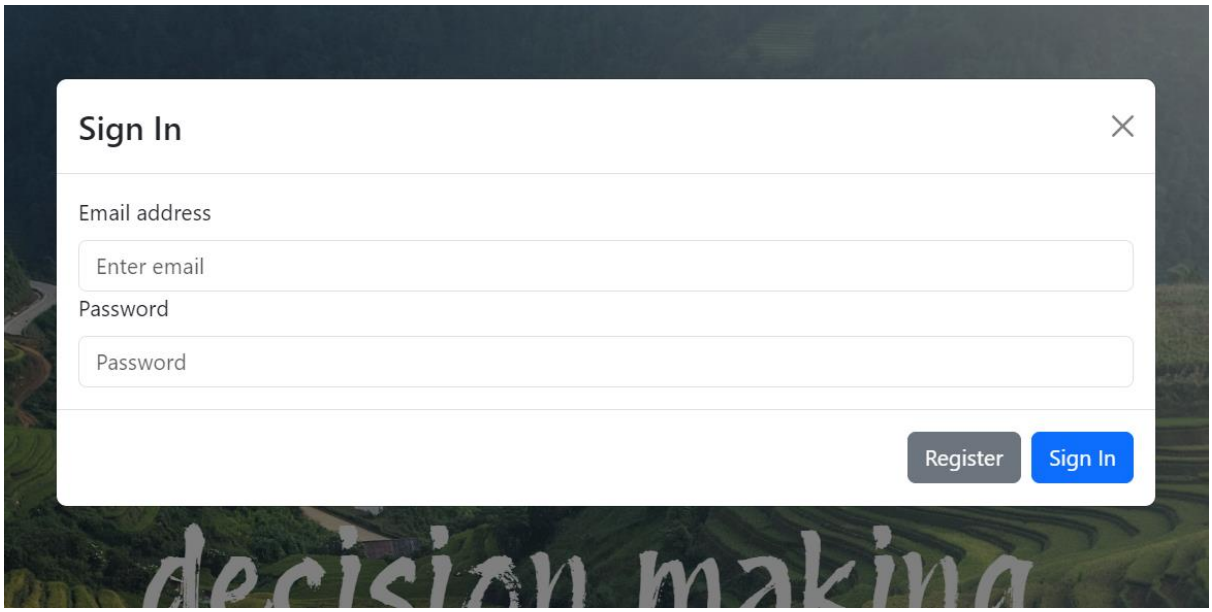
# Appendix L

**React WebApp UIs : Here some UIs are showcased,**

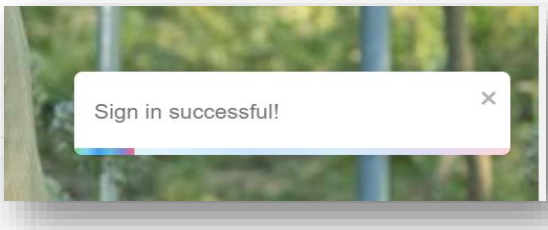
Admin, Agriculture field officer, farmer etc. particular views and operations within the system

**Home Page**

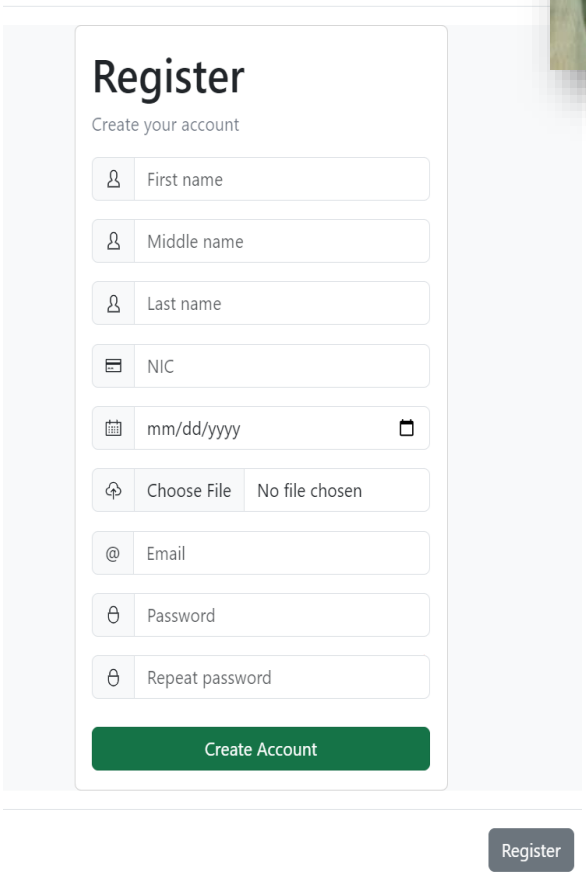
**SignIn form**



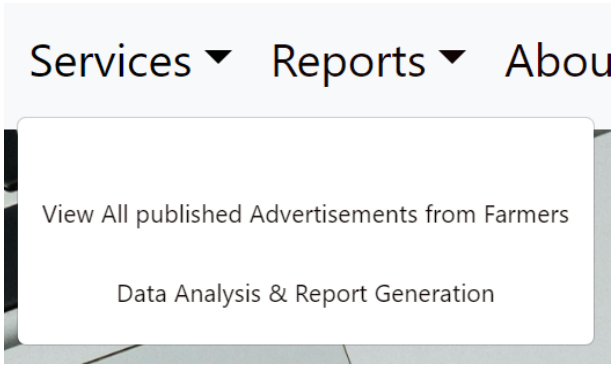

**Register form**

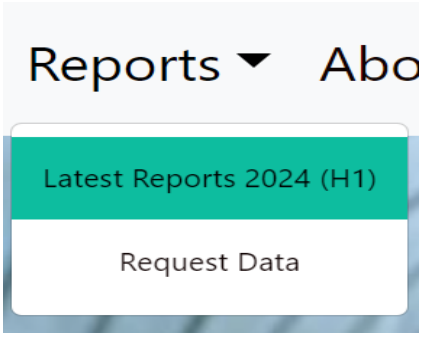



SignIn successful toast message

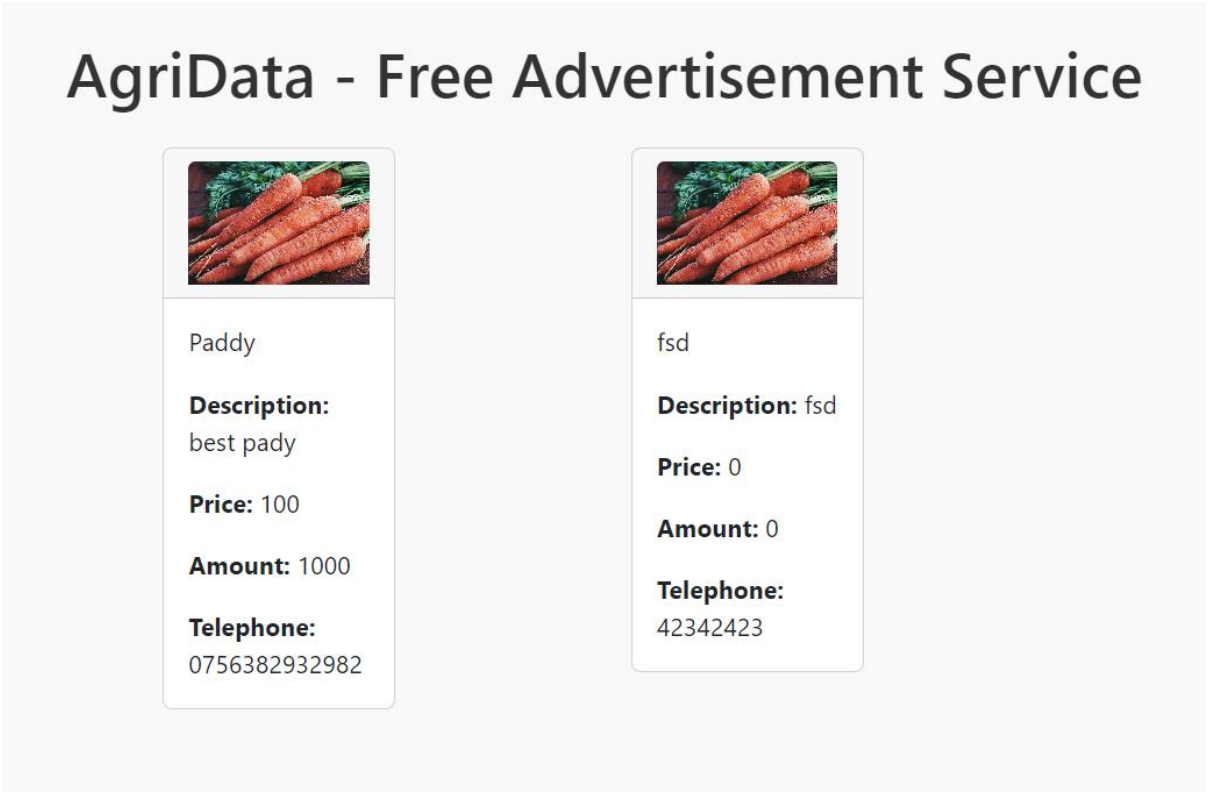


**Nav Bar drop Downs**

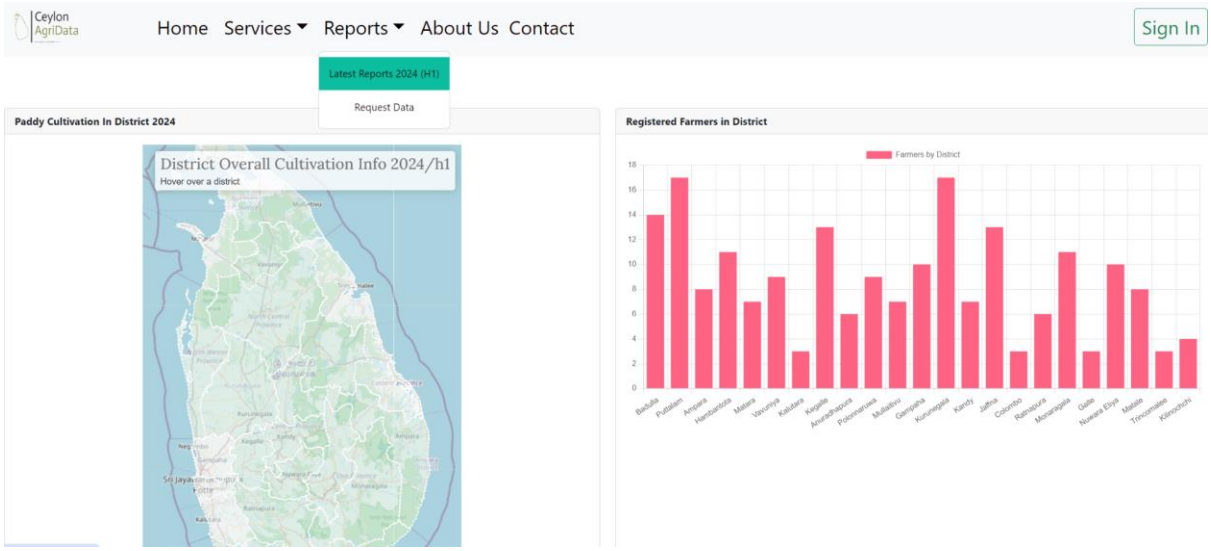



**Free Advertisement Service**



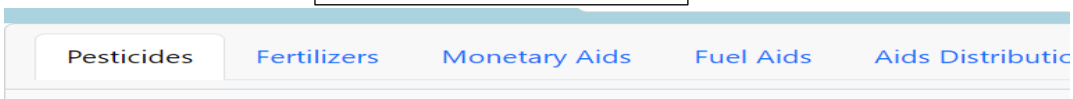
**Publicly available some reports**



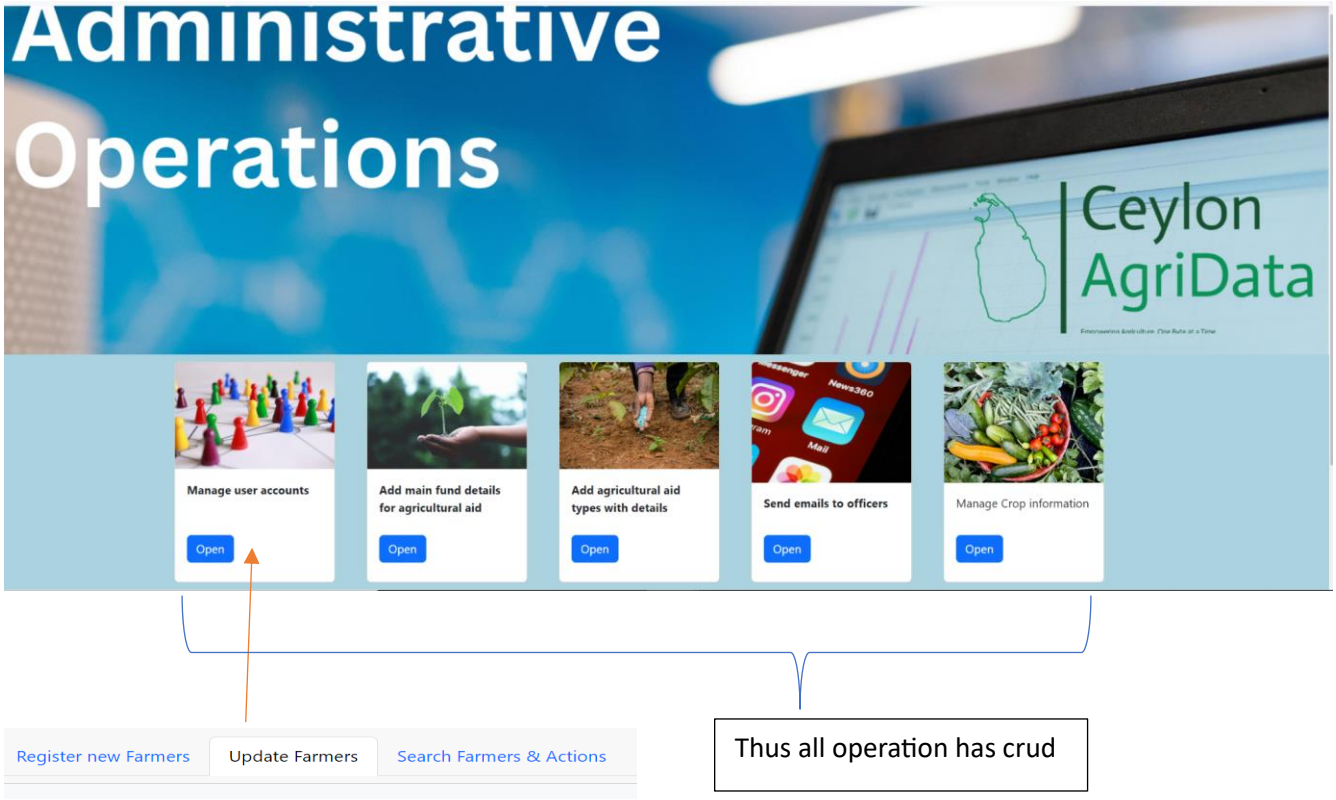

**Admin reports types**

- Total Registered Users
- Estimated Harvest vs Actual
- Monthly Aid Distribution to Farmers
- Agricultural Aid Distribution Funding & Aids
- Crop Yield by Maps
- Field Mapping
- Tax Prayer Report
- Disaster Overview Report
- Advertisement Overview Report

Types of Aids



Admin Operations







Thus all operation has crud

CRUD operation of farmer