ASHESI UNIVERSITY

A RESOURCE PLANNING SOFTWARE FOR YOUTH AGRIPRENEURS IN AFRICA

APPLIED PROJECT

B.Sc. Computer Science

Wuyeh Jobe

2019

ASHESI UNIVERSITY

A RESOURCE PLANNING SOFTWARE FOR YOUTH AGRIPRENEURS IN AFRICA

APPLIED PROJECT

Applied Project submitted to the Department of Computer Science, Ashesi University in partial fulfilment of the requirements for the award of Bachelor of Science degree in Computer Science.

Wuyeh Jobe

April 2019

DECLARATION

I hereby declare that this Applied Project is the result of my own original work and that no part of it has been presented for another degree in this university or elsewhere.

Candidate's Signature:

....

Candidate's Name:

....

Date:

....

I hereby declare that preparation and presentation of this Applied Project were supervised in accordance with the guidelines on supervision of Applied Project laid down by Ashesi University.

Supervisor's Signature:

....

Supervisor's Name:

....

Date:

....

ACKNOWLEDGEMENT

My sincere gratitude to my parents and guardians for supporting me throughout my educational journey. To Mastercard Foundation, I am really honored to be awarded this prestigious scholarship to pursue my undergraduate degree. Thank you. To Ashesi University, I am grateful for all the opportunities and support throughout my stay here. To my supervisor, thank you for the great ideas you shared that made this project a success.

ABSTRACT

Across the continent of Africa, there are young people who use their experience in farming and knowledge in business to engage in agriculture in a whole new way. Unlike traditional farmers, these young people are in tune with technology and are passionate about changing the status quo on the continent. They are called Agripreneurs.

The world bank projects that the world's population will reach about 9.6 billion by 2050 and this means that current food production needs to be increased by 70 per cent. A significant percentage of the increase in the world's population will come from the continent of Africa. The continent will have about 2 billion people at this time. The role of Agripreneurs in improving food security on the continent will be paramount.

This project is about creating a resource planning software for youth Agripreneurs on the continent to help them manage their resources in the face of disastrous land tenure systems, climate change, and the increase in the use of land for non-agricultural purposes.

Table of Content

Chapter 1: Introduction

1.1 Background

Technology has played a huge role and will continue to play a huge role in agriculture. Even the pundits that argue that the problem with our food security systems is not one of production, but that of distribution [2] acknowledge that technology's role in the agriculture value chain is indispensable. Some researchers suggest that there are opportunities to create technologies that can assist rural farmers in their daily farming activities [3]. If technology can make farming easier for rural farmers who, generally, are not tech-savvy, imagine what it can do for youth farmers who have been exposed to various forms of technology.

Indeed, there have been various forms of technology designed to support farmers and agripreneurs alike. Whether it is a platform for sharing information or for connecting them to markets [7], these technologies have shaped the ways youths and farmers, in general, engage in Agriculture. It is easy today to notice the importance of these technologies across several parts of the continent. UjuziKilimo [21] in Kenya leverages the power of big data to provide meaningful information to smallholder farmers. Zenvus [8] in Nigeria uses precision agriculture techniques to measure and analyse soil properties to advice farmers to apply the right amount of fertilizers. Famerline [22] in Ghana provides a "market-driven solution that empowers over 200000 farmers." Agrospaces [23] in Cameroon provides farming advice and connect farmers to markets. The list goes on.

1.2 Motivation

For long, agriculture, which is the foundation of food security systems across the continent, has been entrusted to the old and the feeble of our communities. In contrast, many youths in search of ‘greener pastures,’ dominate almost all other sectors of our economies. This unfortunate reality is based on the notion that agriculture is for the poor and the destitute of our communities.

The World Bank projects that the world’s population will reach about 9 billion by 2050, which means that current food production needs to be stepped up by 70 per cent [1]. Scarier is the fact that a significant percentage of this increase will come from the African continent, which will have 2 billion people by 2050 [8].

With the massive increase in populations and the resulting high demand for food, there is the need to put youths at the centre of the struggle for food security. More so, there is the need to equip these youths with skills and resources to be productive even in the face of barriers such as lousy land tenure systems, climate change resulting in floods and droughts, and the increase in the demand for land for non-agricultural purposes.

1.3 Problem description

For the most part, the technologies that have been developed for the African continent are targeted to smallholder rural farmers that have very limited access to information and not in tune with the power of technology. However, there is “a new breed of young entrepreneurs combining their love of farming and agriculture with an acquired professional business approach” [6]. These people are called Agripreneurs. These young people, unlike traditional farmers, are in tune with technology and they have an unwavering passion for changing the status quo on the continent. They need to be empowered because they have the zeal and know-how to tackle the problems that agriculture sector on continent presents.

Unfortunately, regardless of the vastness of the technologies provided today, a vital technology remains missing for these agripreneurs – a resource planning software system. This is not to say that resource planning systems are inexistent. There are so many of them. But these systems are not designed for people who engaged in agriculture on a small scale and in most cases, not for farmers on the continent [8]. They are designed for large corporations who often pay exorbitant fees to use these systems. Consequently, farmers who want to include technology in their activities face financial challenges and find foreign farm technologies difficult to operate [8].

Nonetheless, a resource planning software system is essential. It can help reduce operational expenses, improve productivity, provide readily available information, and increase access to accurate data with ease [4,5].

1.4 Project Overview and System Approach

The goal of this project is to create a resource planning software system for youth Agripreneurs on the African continent. This resource planning system will have features to manage planting, watering, harvesting, sales, payroll, schedules of operations, broadcasting of announcements, sending messages, and calculating water and fertiliser requirements of a farm. Most of these will have data analytics functions to provide at-a-glance reports of the various activities that take place on their farms. This system is crucial because it will enable them to manage the operations on their farms and collect data (e.g. record of planting and post-planting activities; farming inputs; production and sales volume, etc.) to make informed decisions in the future and improve productivity. With these reports, the Agripreneurs can make sense out of the data and see trends to make informed decisions. This system will also contribute to the digitization of agricultural information to help upcoming youth farmers learn and be more prepared to effectively tackle the challenges in the agriculture sector.

1.5 Related work and Existing Solutions

As established already, different technologies have been built to enhance agriculture in different parts of the world including the continent of Africa. From simple technologies to complex and intelligent ones, these technologies have changed the way people engaged in agriculture. With the advent of artificial intelligence and the availability of "big data," there has been an increase in the number of technologies that enhance precision agriculture and smart farming [25]. Yet, for the most part, the usage of these technologies on the African continent is low mostly because of the unavailability of data and the lack of knowhow to operate the technologies. However, the African Union in a recent report titled "Drones on the Horizon: Transforming Africa's Agriculture System," argued that despite these challenges, drones can be used to enhance the agriculture sector of African countries [26]. But even with drones, they highlighted that complimentary technologies such as data storage and analytics software would be indispensable [26]. This is a compelling reason for why an Enterprise Resource Planning Software (ERP) is necessary for farming on the African continent.

El Mohadab et al. [4] extensively discussed ERP as a software system "that manages and track all information and operational service in a company." They contrasted open source and paid ERPs that offer a wide range of solutions to small, medium and large companies. By giving examples of existing ERPs such as Odoo, Open Bravo 3.0 and Dolibarr, they identified how both open source and paid ERPs are used by businesses to automate resource planning [4]. What the paper reveals is that paid ERPs offered by vendors such as Oracle, SAP, and Microsoft are very expensive, and thus small businesses (the category in which many youth agribusinesses belong) cannot afford their services [4].

One of the most popular ERPs for agriculture is FarmERP [24]. It offers a software solution for farms and farmers to help them effectively manage their activities. It seeks to promote digital farming, smart farming, and data-driven farming. The software is created in India but has a global customer base including some Africa countries. Although it claims it has an app for small farms called FarmERP lite, most of its customers are large agricultural enterprises.

Chapter 2 - Requirements

2.1 Requirement Gathering

Gathering the requirements for the functionalities of this system comes in two forms: collecting data from potential users of the system and research into already existing ERPs for agriculture. My research into existing ERPs for agriculture reveals that most of them have the following functionalities:

A. Tracking the planting and harvesting history of a farm

B. Recording financial transactions and generating financial statements

C. Water and fertiliser requirement calculators

D. Managing sales of agricultural products

E. Broadcasting announcements

F. Payroll manager

G. A scheduler for agricultural operations which get converted directly to work done list

H. Livestock management system

After this general search, a survey was sent to people who are engaged in various aspects of the agriculture value chain and people who are interested in the field of agriculture. In the survey, seven main questions were asked, and they are as follows:

1. On a scale of 0-5, where 0 means not necessary and 5 means extremely necessary, what do you think about creating a system like this?
2. Explain your choice from question 1 above

3. Have you used technology in any of your activities? (If yes) What technologies did you use and what did you like about them? (If no) Why?
4. Generally, what do you think about technologies that seek to improve agriculture?
5. If this system was to be customized for your needs, what features must it have? Select all that apply to you (I listed the functionalities above that I found from my online research)
6. List all other needs not captured above
7. Any other information you want to provide that you think may be helpful for building this system?

The goal of the survey was to validate these requirements and get as much information as possible. Ten people participated in the survey: four from Uganda, three from Ghana and the remaining three from Kenya, Rwanda and Zimbabwe. Eight (constituting 80%) of the participants think that the system is very necessary (i.e. 5 out 5), and the remaining two gave a score of three out of five. At least, five people believe that all the functionalities are necessary. Some functionalities such as (A) have a maximum of eight people believing that they are essential. For this capstone project, however, I am not going to implement the financial management tool (B) and livestock management system (H) listed above. Implementing these two requires more time than this capstone project provides, and there are open source software for them. I am going to recommend some of them for the Agripreneurs.

2.2 Functional Requirements

Below are the key functionalities I am going to implement:

1) Tracking the planting and harvesting history of a farm

2) Water and fertiliser requirement calculators

3) Managing production (planting, watering, harvesting) and sales of agricultural products

4) Broadcasting announcements

5) A messaging system

6) Payroll manager

7) A scheduler for agricultural operations which get converted directly to work done list

8) Assigning Roles and Functionalities that users should access

9) Authentication

10) A backup system

11) Generating and sharing data

The requirements for each of the above functionalities are listed below:

1) Tracking the planting and harvesting history of a farm

a) A user of this functionality should be able to record planting details (crop, variety, date/time planted, plot location, planting type - seed/transplant, seed source/nursery, amount seeded, amount germinated, seed rate/population, bloom/fruit date, volume of water needed, plant traits)[1]

b) A user should be able to record harvesting details (date, crop, plot location, yield, labour hours, storage area/method, notes)[1]

c) A user should be able to update planting and harvesting details

[1] 7 Crop Record-Keeping Chart Sheets: https://www.hobbyfarms.com/7-crop-record-keeping-charts-5/

d) A user should be able to delete planting and harvesting details

e) A user should be able to view planting and harvesting details

f) The system should allow users to see reports in the form of charts and tables

2) **Water and manure/fertilizer requirement calculators**

   a) A user of this functionality should be able to record soil condition details (date of data collection, plot location, soil observations, soil test results, fertiliser or amendment application, volume, notes)[2]

   b) A user should be able to update/delete soil condition details

   c) The system should be able to calculate the manure/fertilizer requirements based on the collected data

   d) A user should be able to record rainfall/irrigation details (date/time, crop/plot location, rainfall amount, irrigation system used, start time, end time, the volume of water used)

   e) Based on the planting and rainfall/irrigation details, the system should calculate and suggest the volume of water to be used to meet the correct need of a crop/plot

3) **Managing sales of agricultural products**

   a) A user of this functionality should be able to add products (with the details: name, quantity, price, wholesale/retail, image)

   b) A user should be able to delete products

   c) A user should be able to update products

   d) A user should be able to search for products

   e) A user should be able to view products

   f) A user should access a point of sales dashboard to sell to customers and generate a receipts

[2] 7 Crop Record-Keeping Chart Sheets: https://www.hobbyfarms.com/7-crop-record-keeping-charts-5/

4) **Broadcasting announcements**
   a) The Admin can send announcements to all staffs at the same time
   b) Announcements can be broadcasted with different urgency levels - high, medium and low
   c) All staff should receive timely notifications whenever information is broadcasted by management
   d) Periodic reminders should be sent based on the urgency level
   e) The Admin can decide which staff should be able to broadcast announcements

5) **Messaging System**
   a) An admin user should be able to send private messages to staff
   b) A user should be able to send messages to admin and other users
   c) Users should be able to receive message notifications
   d) Users should be able to attach files when sending messages

6) **Payroll manager**
   a) A user of this functionality should be able to record the breakdown of the payment due to each staff (basic salary, bonus, overtime, deductions, etc.)
   b) A user should be able to delete and update the payment breakdown of each staff
   c) A user should be able to view both individual and the aggregate of the staff's payment details
   d) A user should be able to view his/her payment breakdown

7) **A scheduler for agricultural operations which get converted directly to work done list**
   a) A user should be able to create a schedule for the operations s/he will be doing for each day (date/time, task, expected time to complete, tools/resources needed)
   b) Admin can allocate the operations that each staff will do for a day
   c) All schedules are automatically recorded

d) A user can update the details of operations they set for themselves

e) Only the admin can update the details of the operations they allocate (in this case, an affected staff is notified)

f) If a user needs to update an operation allocated to them, they can send a message to management

8) Assigning Roles and Functionalities that non-admin users should access

a) Admin should be able to create users and assign roles and functionalities that users should access

b) Admin should be able to delete or update information about users.

c) An email should be sent to a user when an admin creates a role for him/her to set up an account

d) Users should only see and access the functionalities that they are assigned to

9) Authentication

a) After a user is created by an admin, the user gets an email with a link to set up an account

b) After an account set up, a user should be able to log into the system

c) A user can request a password change

10) Backup System

a) Admin should be able to backup all data at anytime

b) Admin should be able to view backup history

c) Admin should be able to restore backup whenever the need arises

11) Generating and Sharing data

a) Users can generate CSV/PDF files of the data tables

b) This CSV can be downloaded and shared with people who might need the data for any reason

2.2.1 Users of the System

This system is designed for Agripreneurs who engage in several aspects of agriculture ranging from planting to sales and everything in between. However, these Agripreneurs deal with people with varying roles. From one Agripreneur to another, the roles of their staff differ. More so, sometimes, the roles are not clearly defined. Therefore, it makes sense to give the Agripreneurs the flexibility to assign roles. Because of this flexibility, the use cases for the system are numerous, and the potential roles are numerous. However, below are some roles that may be predominant:

a) **Agripreneurs**: These people will play a major role in administering the system. They are responsible for assigning functionalities that other users should access and teaching them how to use it.

b) **Farmers**: Farmers most often deal with planting, harvesting, watering, irrigation., etc Thus, they are likely going to use most of the functionalities listed above.

c) **Sales Personnel**: These are people who engage in the day to day sales of the products, and they are likely going to use the functionality that helps manage sales.

d) **Extension Workers**: Agriculture extension workers most often are government workers that work with farmers to help them address problems they have. They can use this system to help farmers record information. Base on the information they collect, they can make informed decisions.

e) **Government**: Agriculture ministry/department can request for information from the Agripreneurs for decision making

f) **Developers**: Developers can integrate the system into other systems

2.2.2 Use cases Diagram

Below is are detailed use cases diagrams for the system:

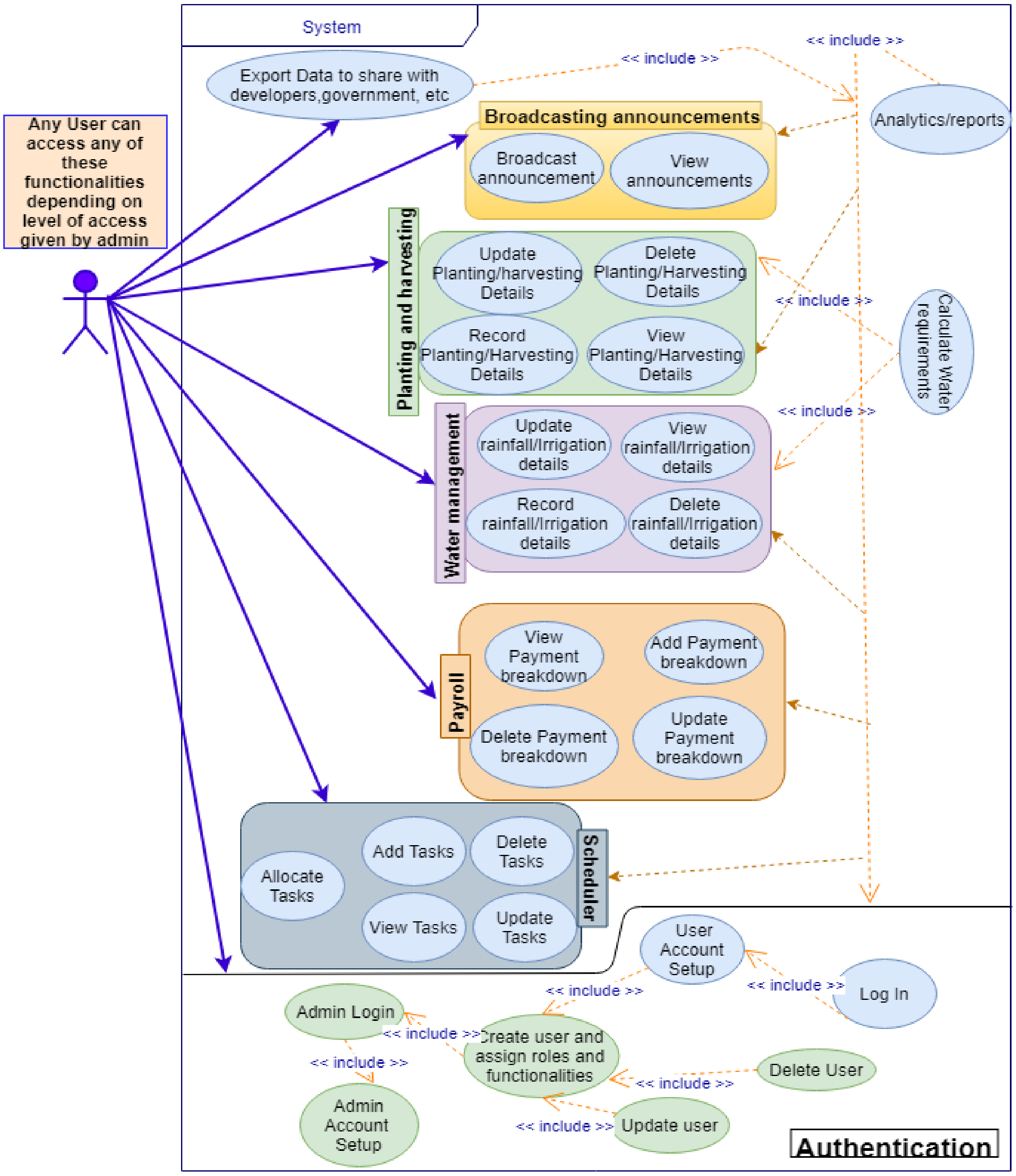

System
Any User can access any of these functionalities depending on level of access given by admin
Export Data to share with developers,government, etc
<< include >>
<< include >>
Analytics/reports
Broadcasting announcements
Broadcast announcement
View announcements
Planting and harvesting
Update Planting/harvesting Details
Delete Planting/Harvesting Details
Record Planting/Harvesting Details
View Planting/Harvesting Details
<< include >>
Calculate Water requirements
<< include >>
Water management
Update rainfall/Irrigation details
View rainfall/Irrigation details
Record rainfall/Irrigation details
Delete rainfall/Irrigation details
Payroll
View Payment breakdown
Add Payment breakdown
Delete Payment breakdown
Update Payment breakdown
Scheduler
Allocate Tasks
Add Tasks
Delete Tasks
View Tasks
Update Tasks
User Account Setup
<< include >>
<< include >>
Log In
Admin Login
<< include >>
Create user and assign roles and functionalities
<< include >>
Admin Account Setup
Delete User
<< include >>
<< include >>
Update user
Authentication

Figure 2.1 Use cases diagram

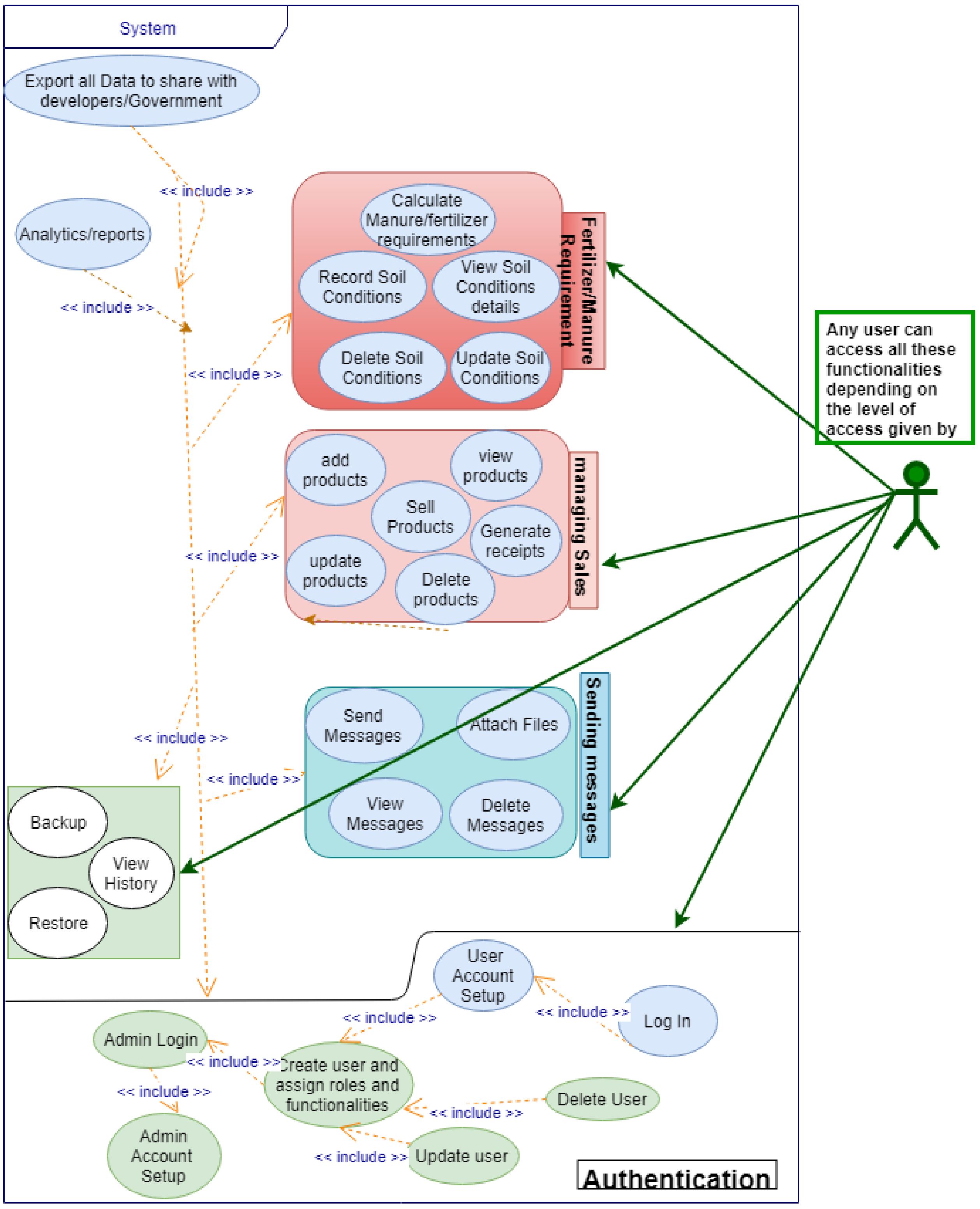


Figure 2.2 Use cases diagram (cont.)

2.2.3 Scenario

Salifu Kunta is an Agripreneur from the Gambia. He and his team work relentlessly every day to cater to the food needs of his community while earning money to support himself and his family. His team comprises of farmers, sales personnel who doubles as an accountant, and a researcher. He also partners with other farmers in the community to assist them in their production activities and make them more productive. From time to time, he gets visits from extension officers from the ministry of agriculture, who come by to provide support in the form of disease prevention in his farm and also interact with the farmers that he partners with. As you can imagine, a lot of things happen in Salifu's farm. Throughout these processes, a lot of data is generated yet there is no efficient system to record this data, much more extract information from the recorded data.

In most cases, such as during sales, the sale personnel records the information in a book. There have been cases when the record book was soaked in rainwater when she forgot the record book on one of the trees. Valuable information was lost. She tried to recollect some of the transactions that occurred from memory but to no avail. This devasting experience is nothing compared to the opportunities lost in not building a database for very valuable data generated from the farm.

Surprisingly, everyone on the farm has a smartphone. Salifu has a computer. Most of the farmers he works with have attained some level of education, and with little support, they would be able to assist in recording data. Salifu needs this system to manage all his activities from planting down to selling his products and everything in between, generate reports to share with other stakeholders, and see analytics to make informed decisions.

2.2.4 Non-Functional Requirements

Security: Since the system helps record valuable information about agribusiness, it is important to secure the system from external attacks. Throughout the coding process, best security practices are employed. The system is built using Python Flask (a Python web framework) which provides an elegant routing system that provides an extra layer of security in accessing pages to be rendered in browsers. Also, when submitting forms such as registration, login forms and all other forms for recording data, the "POST" method is used rather than "GET". This prevents user information from not being displayed as a link in the browser.

When a user is created by an admin, and an email is sent to the user, the system encrypts the ID of the user such that no one will have an idea of how user ID is generated from the database. Also, the link that is sent to the user has an expiry time so that unauthorised people are not able to use the link if in case they access the link in some dubious ways. The same technique is used when a user request for a password change.

For database security and many other reasons (discussed later), Flask SQL Alchemy (which is famously referred to as the Python SQL toolkit) is used as the choice for database interaction. It has features such as 'auto-escaping' that protects databases from SQL injection. I make use of best practices in Flask SQL alchemy to ensure that security is guaranteed.

Availability: Once the system is deployed, it will become part and parcel of the business. This is why it is important to maintain availability at all times (especially during working hours when a lot of operations occur). Since hosting is going to be taken care of by the agribusinesses using the system, recommendations will be provided to ensure the businesses get the most out of the system (i.e. get hosting that provides availability of the system 99.9% during working hours).

As part of availability and security, the application makes it possible to backup the database and be able to restore any version of the backup at any point in time.

Performance: Performance is very important for this system because it is very depended on data. SQL Alchemy provides speed since it is designed specifically for python. With SQL Alchemy, tables in the database are designed as python objects in such a way that class names represent tables and the attributes represent the columns. This makes it very fast to add, delete, update, or retrieve data.

Additionally, it provides flexibility on the choice of the various SQL databases: SQLite, MySQL, and Postgres. What this means is that, during the process of setting up, an Agripreneur can decide to choose any of these databases. For instance, there may be Agripreneurs who will like to store huge amount of information and won't mind getting themselves databases online. In this case, MySQL and Postgres will be the databases to be recommended. On the other hand, users who don't want to be bothered with setting up databases, SQL lite will be the default storage. SQLite is a database that resides in the application as file and can provide almost all features that exist in MySQL and Postgres. However, for agribusinesses who may want to store a huge amount of data, the other two would be highly recommended during the process of setting up.

Scalability: The good thing is that even if users are not sure of the choice of database, they can always go with one that suits them the most (default set up being SQLite). With time, when there is the need to scale up, they can always migrate their data to the new database, without losing data.

Usability: I consider usability very seriously in my implementation. The related work I have looked at showed that most ERPs have poor usability characteristics. That is why this system, although it has so many functionalities, embraces simplicity for the user interface. New technologies are used to ensure that, for the most part, users can do what they want without reloading the page.

Chapter 3 - System Architecture

This system is a web application that has various sub-applications combined. It uses Python Flask for the backend, HTML/CSS and JavaScript for the frontend, and SQL-Alchemy for the database. Two main architectures will be used to provide a high-level overview of the system. These are the ***repository*** and ***the client-server*** architectures.

3.1 Repository Architecture

The repository architecture is used to explain the interaction that happens between the sub-applications and the data that is generated by each application. The project has a main database which all the sub-applications interact with. The diagram below shows the interaction between the sub-applications and the database, the repository for the project:

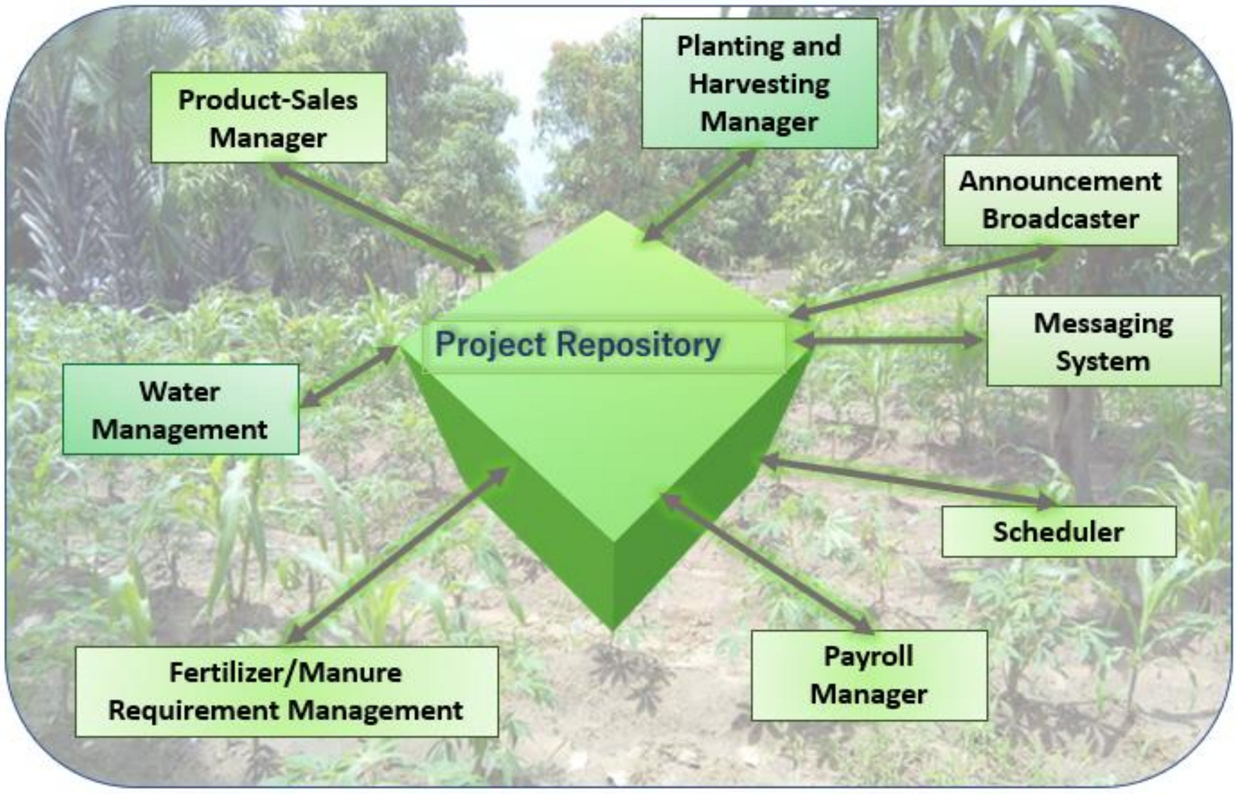


Figure 3.1 The Repository Architecture

Each of the sub-applications generates some kind of data that may be needed by another application. For instance, the Water and Fertilizer/Manure management components require the use of the Planting component. However, rather than the two sub-applications communicating directly, the data from one system is stored in the database and retrieved by another component when the need arises.

From figure 3.1, you can observe that all the various applications (Water + Fertilizer requirement calculator, Production-Sales manager, payroll manager, etc.) are interacting with the database rather than interreacting among themselves. The database is where the information is carefully organized to facilitate easy storage and retrieval of information. Because the usability of this system is largely dependent on the database, it is equipped with a robust backup system.

While the repository architecture indicates the relationships between the applications and their communication with the database, it does not adequately explain how these communications occur during the process of using the application. The client-server architecture seeks to bridge this gap.

3.2 The Client-Server Architecture

The client-server architecture is used because it clearly explains the processes that take place for the system to be used by the clients. The system is hosted on a server and also connected to the database. The clients (web browsers) send requests to the server. The server, which has a connection to the database, will request data from the database. Once the requested data is retrieved, the server sends it to the client, and the process continues.

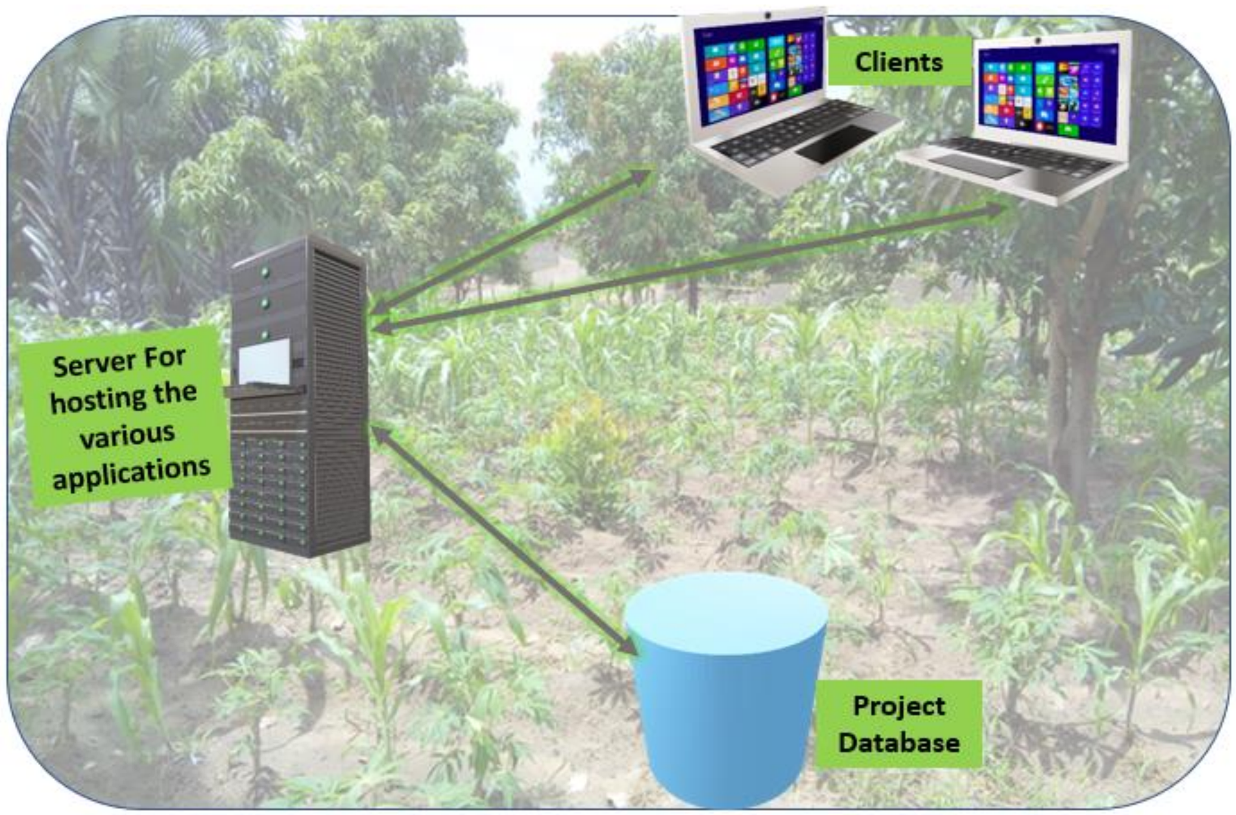


Figure 3.2 The Client-Server Architecture

Figure 3.2 provides a diagrammatic representation of the communication flow between the client, server and the database.

3.3 ER Diagram

From the general description of the system and the architectures discussed above, it is quite clear that a robust database is an integral part of the system. The system makes use of a structured database to store and retrieve data. Below is the ER diagram that shows all the tables for the database and their attributes/columns:

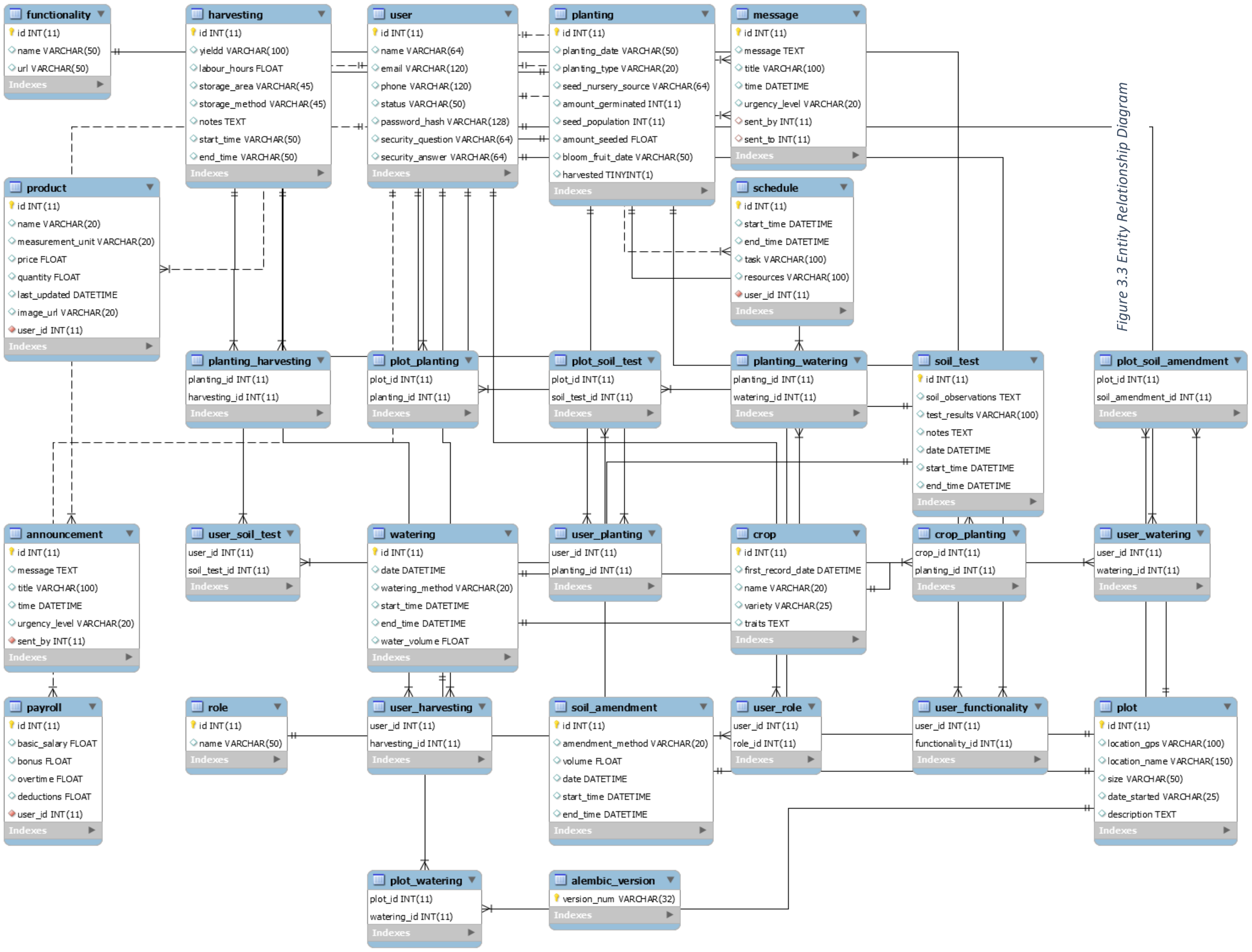


Figure 3.3 Entity Relationship Diagram

Chapter 4 – Implementation

The chapter entails the description of the implementation of the system: the languages and frameworks used for the frontend and backend, the database design and techniques adopted to facilitate the storage and retrieval of data, the mechanisms used to ensure usability and security of the system, as well as the tools used to facilitate the development process.

4.1 Overview

The main language for the development of the backend is the Python web framework called Flask[3]. The languages used for the frontend are Hyper-Text Mark-up Language (HTML), Cascading Style Sheet (CSS), JavaScript, jQuery and AJAX. The frameworks used are Bootstrap and jQuery DataTables[4]. The language used for the database is Flask SQLAlchemy[5]. The tools used include Visual Studio Code, command prompt, XAMPP, git, and GitHub.

4.2 Backend Development

4.2.1 Programming Language - Python Flask

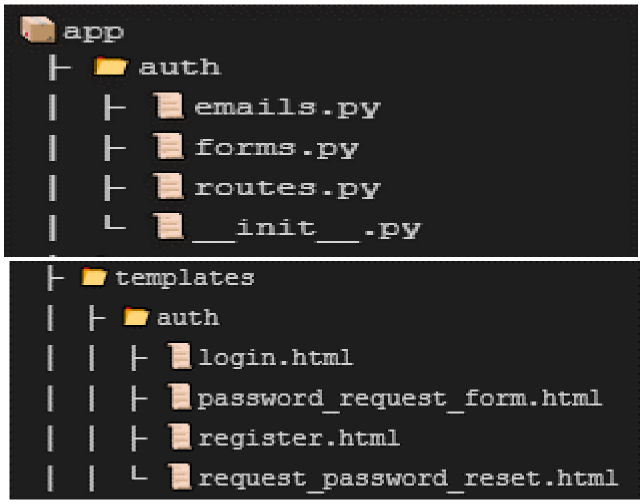
Flask is a Python web framework for building web applications. It is popularly described as "a microframework for Python based on Werkzeug (a Python toolkit for Web server Gateway Interface) and Jinja2 (a templating language for Python)" [10]. The "microframework" means that all the decisions and control rest with the developer. You can decide, for instance, to choose any database you like because Flask does not have a database abstraction layer that restricts you. Its ability to allow developers to integrate extensions and make such extensions as if they were implemented with flask itself makes it even more plausible [11].

[3] http://flask.pocoo.org/
[4] https://datatables.net/
[5] http://flask-sqlalchemy.pocoo.org/2.3/

Flask comes with so many features including a built-in development server and debugger, and support for unit testing. These are some of the reasons why I chose Flask for this project. Most importantly for me, however, I want this software to be integrated with machine learning algorithms to make it more powerful in future. The fact that Python is one of the most popular programming languages used in machine learning and given that Flask provides flexibility and support for such extensions, I am convinced in the choice of backend language for this project.

4.2.2 Application Structure

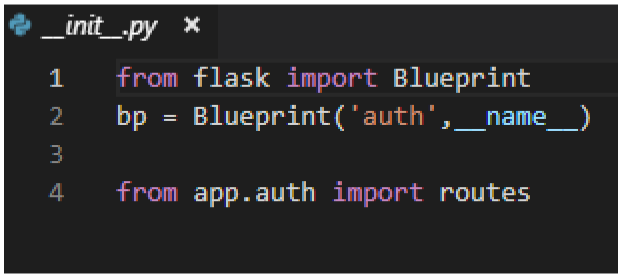
The application structure adopted in building this system makes use of Flask’s blueprint feature. This enables you to develop subsystems and put them together to form the whole system. Flask’s blueprint feature provides convenience (especially during unit testing), efficiency (it facilitates code reuse and maintenance), and a very-easy-to-understand code. Also, with this feature, a subsystem can be activated or deactivated simply by registering or deregistering it with the system [12]. Below is an example of how a blueprint is created and registered with the application.



```
app
├ auth
|  ├ emails.py
|  ├ forms.py
|  ├ routes.py
|  └ __init__.py
├ templates
|  ├ auth
|  |  ├ login.html
|  |  ├ password_request_form.html
|  |  ├ register.html
|  |  └ request_password_reset.html
```

Figure 4.1: Folder and tree for a blueprint

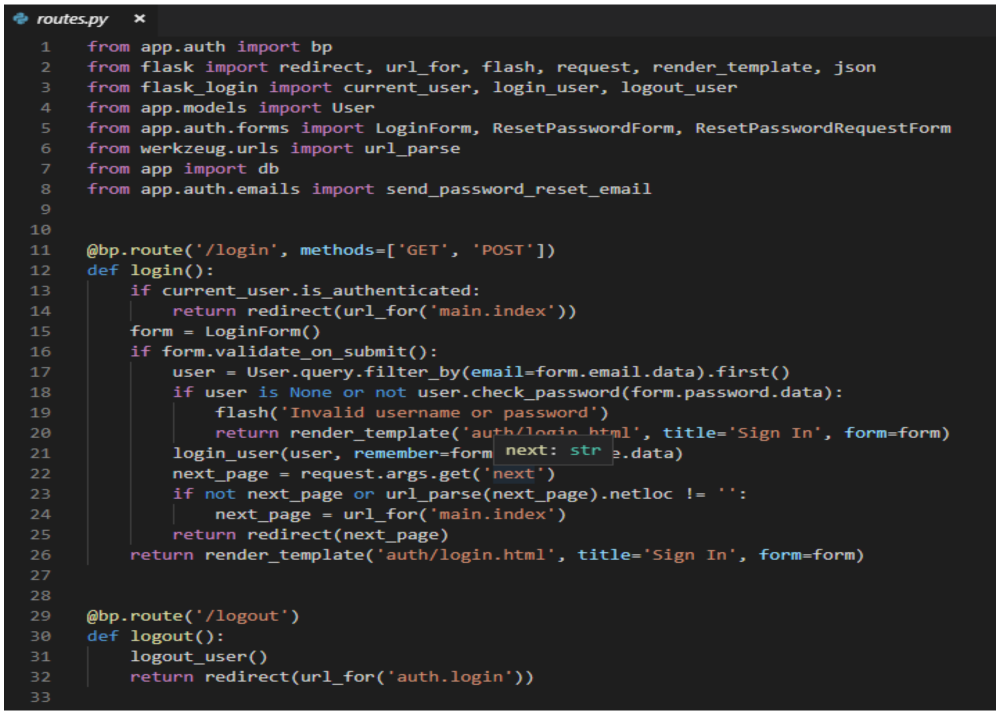
Figure 4.1 shows the arrangement of folders and file for the blueprint set up (see appendix for the complete folder and file tree for the application). The blueprint is declared in the *__init__.py* file and this act as the first point of contact when the application is registered. All the functions of a sub-application are accessed through this file. Observe that the routes (see the explanation for figure 4.3) are imported into this file. Below is the content of the file:



```python
from flask import Blueprint
bp = Blueprint('auth',__name__)

from app.auth import routes
```

Figure 4.2: Creating blueprint

The routes in the application are the gateways to individual functions of a subsystem. They render HTML pages that are specified under them using the ***render_templat*****e** function in Flask. They are also the point of contact for Ajax calls. Below is a diagram showing some content of the *route.py file*.

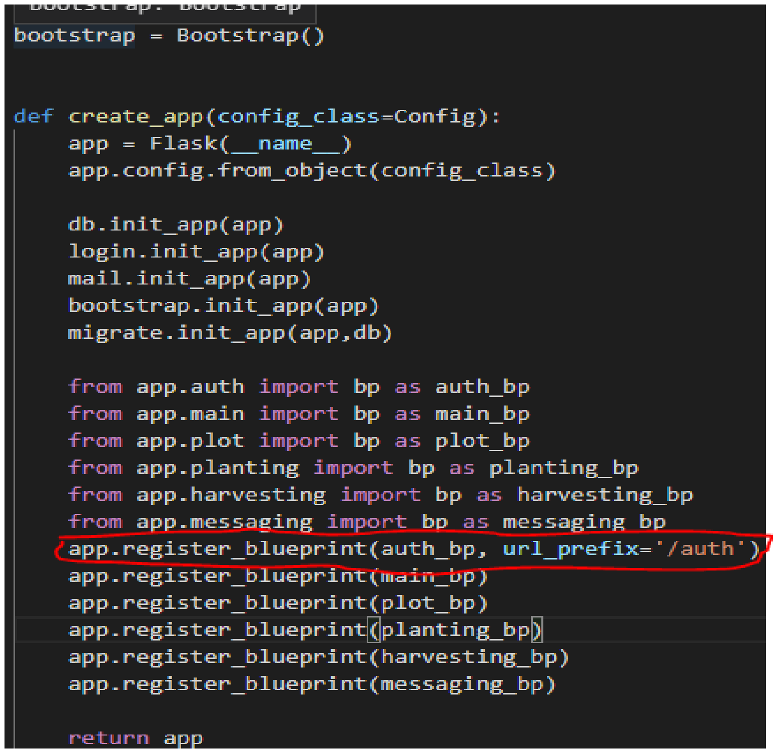

routes.py ×

```python
from app.auth import bp
from flask import redirect, url_for, flash, request, render_template, json
from flask_login import current_user, login_user, logout_user
from app.models import User
from app.auth.forms import LoginForm, ResetPasswordForm, ResetPasswordRequestForm
from werkzeug.urls import url_parse
from app import db
from app.auth.emails import send_password_reset_email


@bp.route('/login', methods=['GET', 'POST'])
def login():
    if current_user.is_authenticated:
        return redirect(url_for('main.index'))
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        if user is None or not user.check_password(form.password.data):
            flash('Invalid username or password')
            return render_template('auth/login.html', title='Sign In', form=form)
        login_user(user, remember=form next: str e.data)
        next_page = request.args.get('next')
        if not next_page or url_parse(next_page).netloc != '':
            next_page = url_for('main.index')
        return redirect(next_page)
    return render_template('auth/login.html', title='Sign In', form=form)


@bp.route('/logout')
def logout():
    logout_user()
    return redirect(url_for('auth.login'))
```

Figure 4.3: route.py file to showcase how routes work

Observe that the final line under the login function renders the login.html file in *templates/auth* (see folder tree in Figure 4.1)

Once this setup is done, the subsystem must be registered with the application. If not, all the routes in the subsystem will not work, which means that the subsystem will not work since the routes are the gateways to the functions of the subsystem. The registration of the subsystem to the application is done in the __init__.py for the main application as shown below:



```
bootstrap = Bootstrap()


def create_app(config_class=Config):
    app = Flask(__name__)
    app.config.from_object(config_class)

    db.init_app(app)
    login.init_app(app)
    mail.init_app(app)
    bootstrap.init_app(app)
    migrate.init_app(app,db)

    from app.auth import bp as auth_bp
    from app.main import bp as main_bp
    from app.plot import bp as plot_bp
    from app.planting import bp as planting_bp
    from app.harvesting import bp as harvesting_bp
    from app.messaging import bp as messaging_bp
    app.register_blueprint(auth_bp, url_prefix='/auth')
    app.register_blueprint(main_bp)
    app.register_blueprint(plot_bp)
    app.register_blueprint(planting_bp)
    app.register_blueprint(harvesting_bp)
    app.register_blueprint(messaging_bp)

    return app
```

Figure 4.4: Registering a blueprint

Once this is done and the application is started, the login page can be accessed from ***domain_name/auth/login***. The "/login" in the link is the name that is passed to the @bp.route function on top of the login function in *routes.py*. The "auth" in the link is due to the ***url_prefix*** parameter that is added when registering the blueprint. So, the rest of the subsystem which does not have a URL prefix can be accessed by just doing ***domain_name/route_for_a_functionality.*** From this point on the login functionality and indeed any functionality for the authentication blueprint is ready! See the screenshot below:

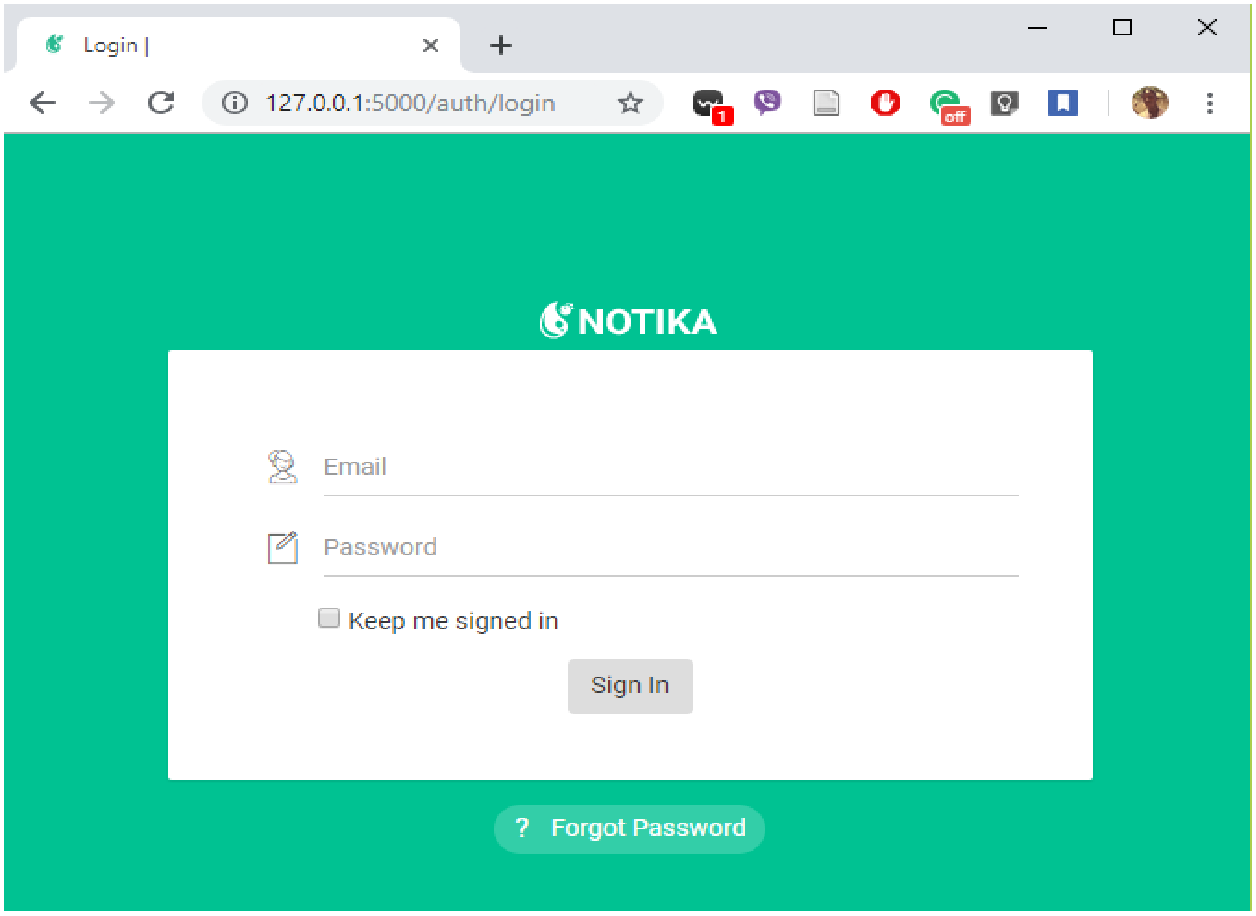


Figure 4.5: Login page ready

All other subsystems are developed in a similar manner. Of course, there is a lot more that has to be done to get the application to work. For instance, getting just the login functionality to work requires importing libraries and modules. In most cases, you have to download Python packages that make it possible to achieve your goals. So, I will admit that the description given here is not complete (it is impossible to detail out all the processes required to get the application to run smoothly). However, understanding this application structure makes it easy to get started and develop remarkable applications in Flask. A course like the Flask Mega-Tutorial [12] provides a holistic tutorial on developing with Flask.

To say that this application structure is befitting for this project would be an understatement. Firstly, it gives me the opportunity to develop the subsystems incrementally and test them effectively. Secondly, it provides the opportunity to detect bugs quickly and enable maintenance of the system once it goes into production. Thirdly it makes it possible to integrate this system with other technologies easily.

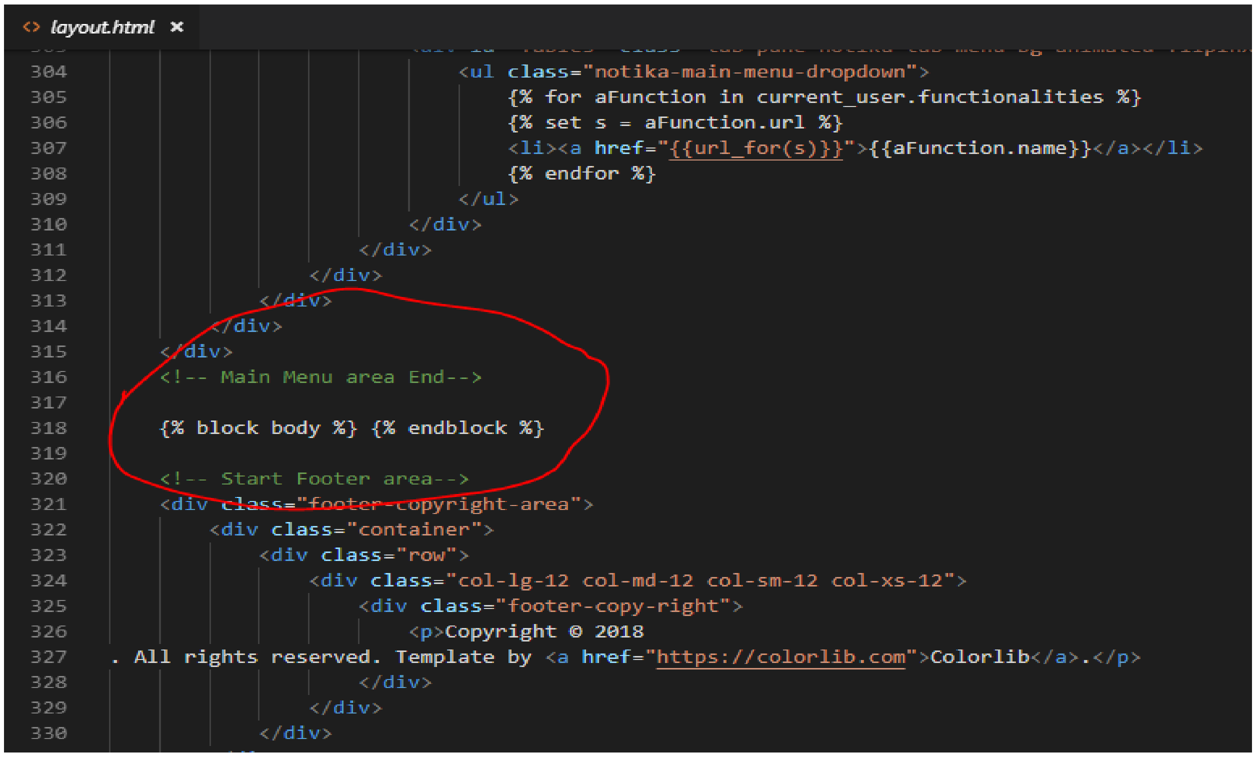
4.3 Frontend Development

4.3.1 Choosing a theme (rather than developing frontend from scratch)

From the onset, the main focus for me in this project was to create software that will be very easy to use and does tasks efficiently. Thus, I had to make a tough decision regarding the choice of interface for the application. Initially, I wanted to build the interface from scratch but realised that was going to take a lot of time, and I may not do a lot of the backend. Also, it will defeat the purpose of code reuse concept in software engineering when I build a frontend from scratch when there are already existing templates that may even better than what I want to build. Taking all these into consideration, I settled on using a Bootstrap (an opensource framework based on HTML and CSS for designing web interfaces, makes it easy to design web interfaces that are responsive to different platforms) template called Notika. It is a free Bootstrap theme that can be used for web applications [13] and it is licensed under The MIT License (MIT) [14]. The template comes with the DataTable jQuery framework, which provides an elegant way of displaying data in tables and gives the opportunity to search and sort data on the client side without having to query from the database. Thus, once the data is loaded, a powerful real-time searching tool is made available to users. This is especially important for this application because it requires the storage of huge amount of data. And this data may be updated and deleted at any time. Trying to search for data to update or delete can be cumbersome if such a functionality is not available.

4.3.2 Customising Template for Flask

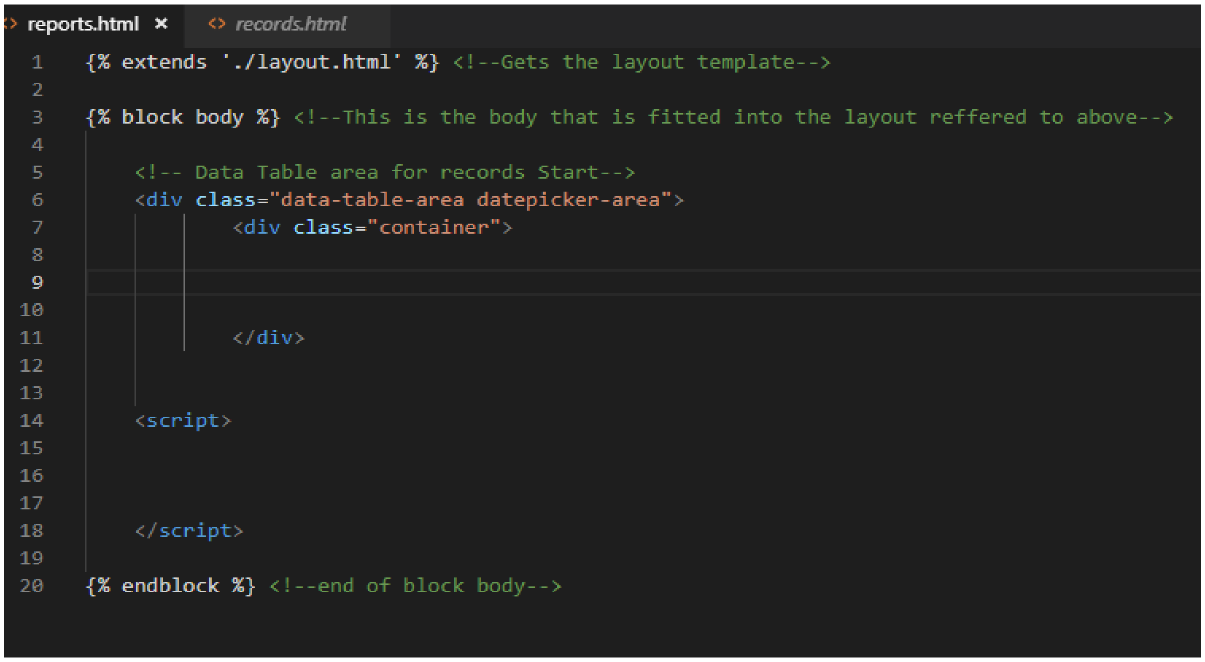
Of course, this template does not come ready to be used with Python Flask. I had to customise it to suit my needs. Firstly, I modularised the code so that I do not have to be repeating code for interfaces that cut across all the pages for the application. I designed a general layout that can be used by all other pages. The layout contains the header and the footer. It makes way for each page to extend it through the Jinja2 block embedded in the page as follows:



```
                            <ul class="notika-main-menu-dropdown">
                                {% for aFunction in current_user.functionalities %}
                                {% set s = aFunction.url %}
                                <li><a href="{{url_for(s)}}">{{aFunction.name}}</a></li>
                                {% endfor %}
                            </ul>
                        </div>
                    </div>
                </div>
            </div>
        </div>
    </div>
    <!-- Main Menu area End-->

    {% block body %} {% endblock %}

    <!-- Start Footer area-->
    <div class="footer-copyright-area">
        <div class="container">
            <div class="row">
                <div class="col-lg-12 col-md-12 col-sm-12 col-xs-12">
                    <div class="footer-copy-right">
                        <p>Copyright © 2018
. All rights reserved. Template by <a href="https://colorlib.com">Colorlib</a>.</p>
                    </div>
                </div>
            </div>
```

Figure 4.6: Layout customized for other pages

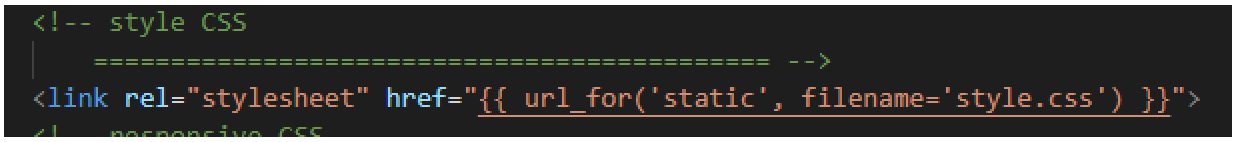
A page seeking to use this layout will simply extend this layout and embed the name of the block in the layout. Any code entered between the block will be displayed between the header and the footer for the layout. Figure 4.7 below illustrates this:



```
reports.html ×    records.html
{% extends './layout.html' %} <!--Gets the layout template-->

{% block body %} <!--This is the body that is fitted into the layout reffered to above-->

    <!-- Data Table area for records Start-->
    <div class="data-table-area datepicker-area">
            <div class="container">



            </div>


    <script>



    </script>

{% endblock %} <!--end of block body-->
```

Figure 4.7: How a page uses the layout

In Flask, all HTML pages are put in ***templates*** folder (refer to folder tree in appendix to see how the template folder is structured). All other resources for the pages are placed in the ***static*** folder (also see the folder tree in the appendix for its structure). Linking a resource to a page is demonstrated in the diagram below:



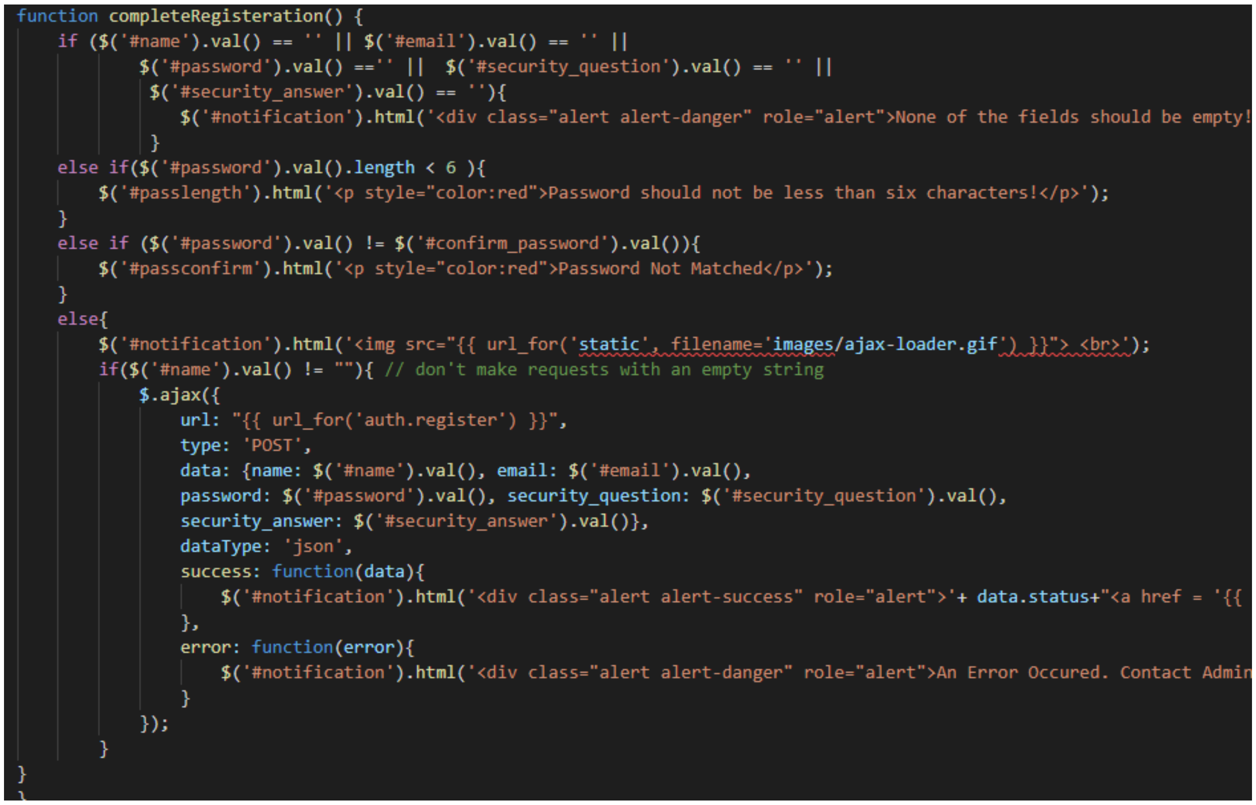
```
<!-- style CSS
    ============================================ -->
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
```

Figure 4.8: Accessing and using a resource

Instead of putting the relative path of the resource as it is done in most programming languages, Flask uses ***url_for*** to establish and generate the path to a resource and access it accordingly.

4.3.3 JavaScript

I used raw JavaScript mainly for validation of inputs from the user and give timely feedback to users when interacting with the system. For instance, when registering, JavaScript is used to compare if the confirmation password matches as the user types the input. JavaScript is also used in some cases to create dynamic elements on a page and also hide or show elements base on user interaction. Below is a sample code to ensure user inputs details before a request is sent to the server through AJAX. The reasons why I did this, although I could have done server-side validation, is that I don't want to send a request to the server unnecessary without correct data.



```
function completeRegisteration() {
    if ($('#name').val() == '' || $('#email').val() == '' ||
            $('#password').val() =='' ||  $('#security_question').val() == '' ||
             $('#security_answer').val() == ''){
                $('#notification').html('<div class="alert alert-danger" role="alert">None of the fields should be empty!
             }
    else if($('#password').val().length < 6 ){
        $('#passlength').html('<p style="color:red">Password should not be less than six characters!</p>');
    }
    else if ($('#password').val() != $('#confirm_password').val()){
        $('#passconfirm').html('<p style="color:red">Password Not Matched</p>');
    }
    else{
        $('#notification').html('<img src="{{ url_for('static', filename='images/ajax-loader.gif') }}"> <br>');
        if($('#name').val() != ""){ // don't make requests with an empty string
            $.ajax({
                url: "{{ url_for('auth.register') }}",
                type: 'POST',
                data: {name: $('#name').val(), email: $('#email').val(),
                password: $('#password').val(), security_question: $('#security_question').val(),
                security_answer: $('#security_answer').val()},
                dataType: 'json',
                success: function(data){
                    $('#notification').html('<div class="alert alert-success" role="alert">'+ data.status+"<a href = '{{
                },
                error: function(error){
                    $('#notification').html('<div class="alert alert-danger" role="alert">An Error Occured. Contact Admin
                }
            });
        }
}
```

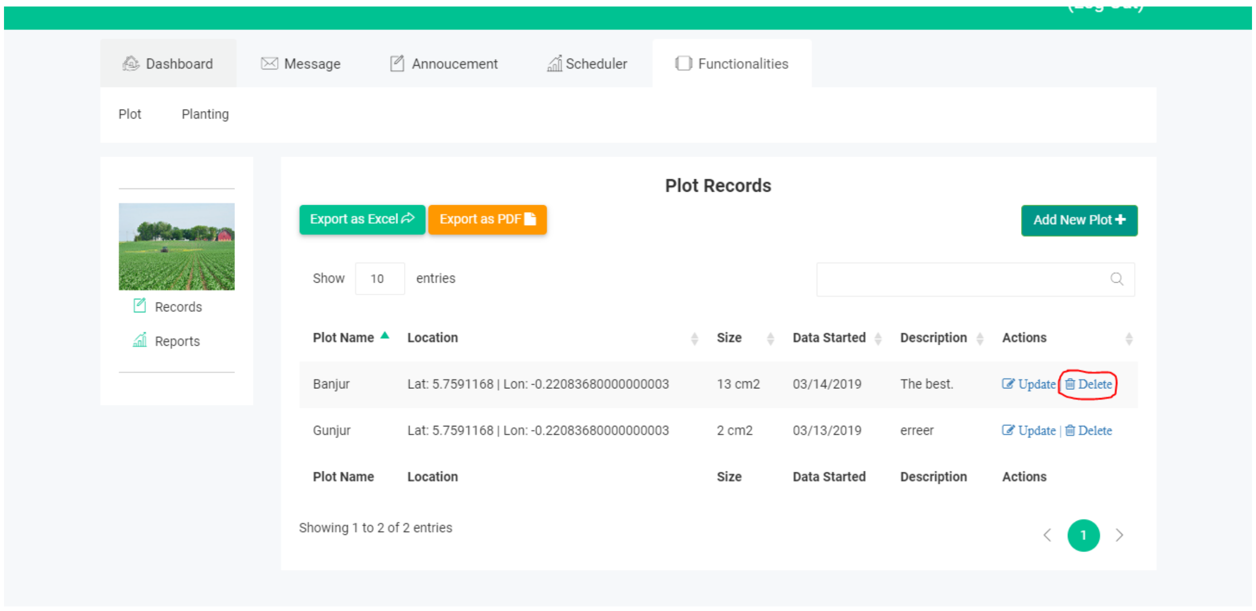
Figure 4.9: Using JavaScript for validation

4.3.4 AJAX

Ajax, which means Asynchronous JavaScript and XML, is a programming technique that makes it possible for clients to send requests and receive responses from a server/database without hampering the user experience. With Ajax, 'users can benefit from seeing certain information generated without having to reload the page' [15]. Ajax is used in almost all the subsystems to execute functionalities like adding, updating and deleting data, sending messages, providing real-time notifications, dynamically updating page content, etc.

Ajax has a profound implication on the usability and performance of the system. It means the page resources are not loaded each time a user wants to use functionalities like these. If a page displays data from the database, it means a query is executed once and all the functionalities can work amicably without reloading the page.

Additionally, Ajax works with Datatables very well. For instance, once a request is sent to delete data from the database and the delete is successful, the data is simply removed from the table without having to do any complicated manipulations to the table. The same thing happens when a data row is updated and, in some cases, when data is added. The process below shows how AJAX is used to send a request to delete the records for a Plot.



Plot Records

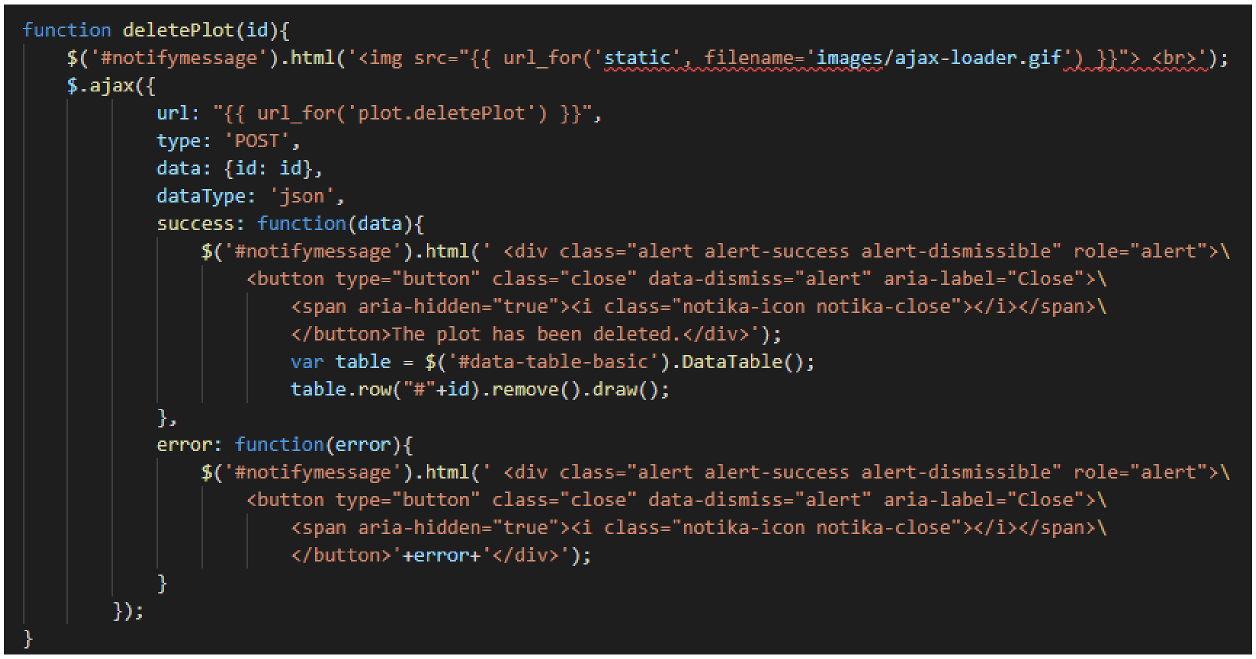
Show 10 entries

Plot Name	Location	Size	Data Started	Description	Actions
Banjur	Lat: 5.7591168 | Lon: -0.22083680000000003	13 cm2	03/14/2019	The best.	Update | Delete
Gunjur	Lat: 5.7591168 | Lon: -0.22083680000000003	2 cm2	03/13/2019	erreer	Update | Delete
Plot Name	Location	Size	Data Started	Description	Actions

Showing 1 to 2 of 2 entries

Figure 4.10: Delete in action

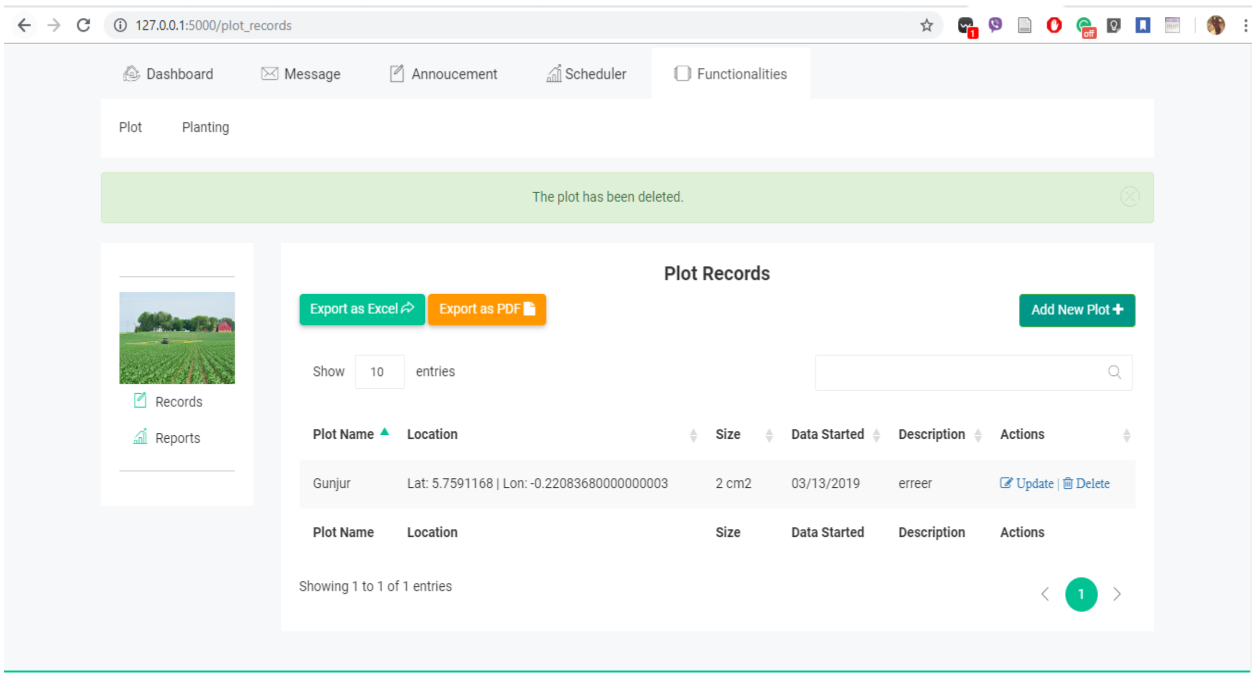
When the “Delete” button is clicked, and the user confirms the delete, the AJAX code below sends a request to delete the record for the plot named “Banjur”.



```
function deletePlot(id){
    $('#notifymessage').html('<img src="{{ url_for('static', filename='images/ajax-loader.gif') }}"> <br>');
    $.ajax({
            url: "{{ url_for('plot.deletePlot') }}",
            type: 'POST',
            data: {id: id},
            dataType: 'json',
            success: function(data){
                $('#notifymessage').html(' <div class="alert alert-success alert-dismissible" role="alert">\
                    <button type="button" class="close" data-dismiss="alert" aria-label="Close">\
                        <span aria-hidden="true"><i class="notika-icon notika-close"></i></span>\
                        </button>The plot has been deleted.</div>');
                        var table = $('#data-table-basic').DataTable();
                        table.row("#"+id).remove().draw();
            },
            error: function(error){
                $('#notifymessage').html(' <div class="alert alert-success alert-dismissible" role="alert">\
                    <button type="button" class="close" data-dismiss="alert" aria-label="Close">\
                        <span aria-hidden="true"><i class="notika-icon notika-close"></i></span>\
                        </button>'+error+'</div>');
            }
        });
}
```

Figure 4.11: Sending request to server using Ajax

Notice how, after a request is successful, the plot is simply removed from the Datatable, as indicated in figure 4.12 below:



The plot has been deleted.

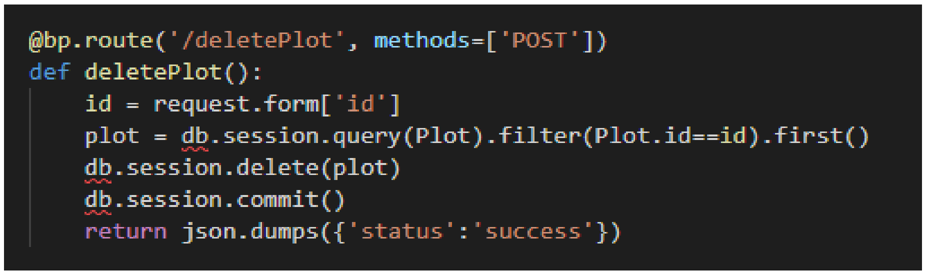
Plot Records

Plot Name	Location	Size	Data Started	Description	Actions
Gunjur	Lat: 5.7591168 | Lon: -0.22083680000000003	2 cm2	03/13/2019	ereer	Update | Delete
Plot Name	Location	Size	Data Started	Description	Actions

Showing 1 to 1 of 1 entries

Figure 4.12: After deleting a plot

Typically, when a request is being sent to the server, four parameters are specified: the *URL* (which in this case represents the route that entails the process for carrying out a certain functionality); *type* (which represent the type of request to make, whether “Get” or “Post”); data (which represents the data to be passed, which could also be data from inputs); and the *dataType* (which represent the data format expected after a successful request). Just for the sake of completing the process of the delete function, below is the route that deletes plot named “Banjur” from the database.



```
@bp.route('/deletePlot', methods=['POST'])
def deletePlot():
    id = request.form['id']
    plot = db.session.query(Plot).filter(Plot.id==id).first()
    db.session.delete(plot)
    db.session.commit()
    return json.dumps({'status':'success'})
```

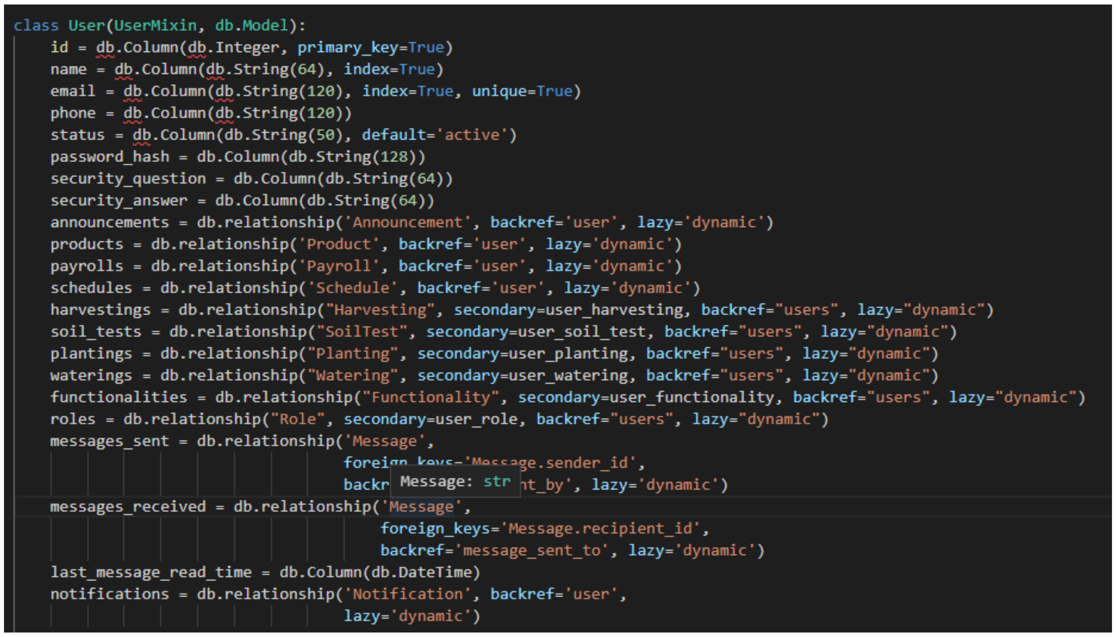
Figure 4.13: Code snippet that delete plot from database using SQLAlchemy

4.4 Database Development

4.4.1 Flask SQLAlchemy

SQLAlchemy is Python toolkit and a powerful tool for performing CRUD (create, read, update, delete) SQL operations without having to write raw SQL statements. It is an Object Relational Mapper (ORM), which means that you can design a database in the famous object-oriented style of programming (using classes, methods, instance variables, etc.) and the object parameters are mapped to the tables of a relational database management system. Flask SQLAlchemy provides SQLAlchemy extension for flask [16]. The advantages that Flask SQLAlchemy provides cannot be overemphasized. As explained in the section for non-functional requirements, it offers flexibility, performance and security. Most importantly though, it offers an elegant way of performing operations without writing lengthy and cumbersome SQL queries.

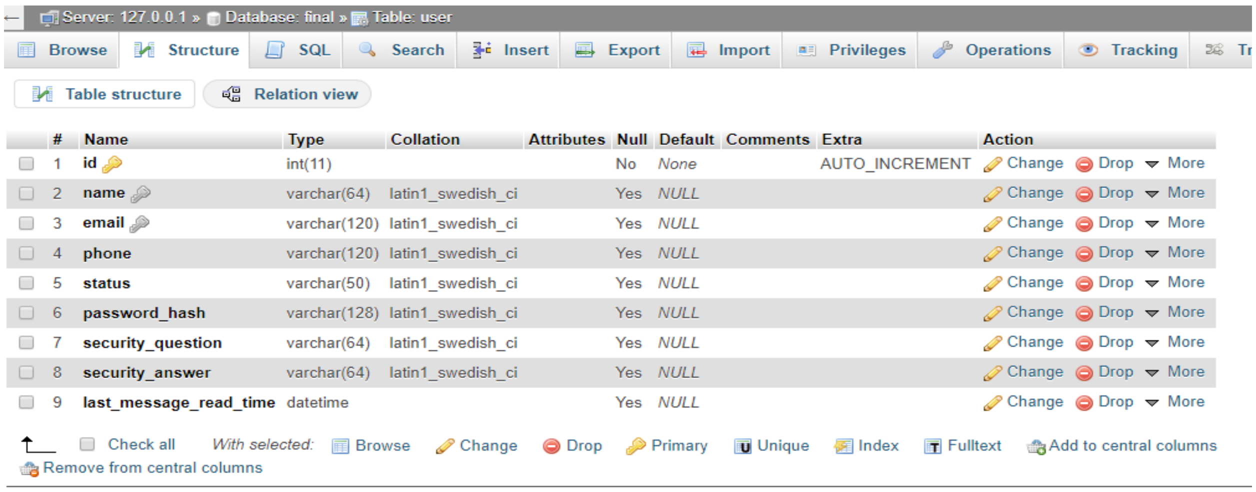
In the example below, I illustrate how Flask SQL-Alchemy is used to create database tables:



```
class User(UserMixin, db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), index=True)
    email = db.Column(db.String(120), index=True, unique=True)
    phone = db.Column(db.String(120))
    status = db.Column(db.String(50), default='active')
    password_hash = db.Column(db.String(128))
    security_question = db.Column(db.String(64))
    security_answer = db.Column(db.String(64))
    announcements = db.relationship('Announcement', backref='user', lazy='dynamic')
    products = db.relationship('Product', backref='user', lazy='dynamic')
    payrolls = db.relationship('Payroll', backref='user', lazy='dynamic')
    schedules = db.relationship('Schedule', backref='user', lazy='dynamic')
    harvestings = db.relationship("Harvesting", secondary=user_harvesting, backref="users", lazy="dynamic")
    soil_tests = db.relationship("SoilTest", secondary=user_soil_test, backref="users", lazy="dynamic")
    plantings = db.relationship("Planting", secondary=user_planting, backref="users", lazy="dynamic")
    waterings = db.relationship("Watering", secondary=user_watering, backref="users", lazy="dynamic")
    functionalities = db.relationship("Functionality", secondary=user_functionality, backref="users", lazy="dynamic")
    roles = db.relationship("Role", secondary=user_role, backref="users", lazy="dynamic")
    messages_sent = db.relationship('Message',
                                    foreign_keys='Message.sender_id',
                                    backr Message: str nt_by', lazy='dynamic')
    messages_received = db.relationship('Message',
                                        foreign_keys='Message.recipient_id',
                                        backref='message_sent_to', lazy='dynamic')
    last_message_read_time = db.Column(db.DateTime)
    notifications = db.relationship('Notification', backref='user',
                                    lazy='dynamic')
```

Figure 4.14: A User class in SQLAlchemy corresponding to a user table

From figure 4.14 above, I created a "User" class. The lowercase of the name of the class corresponds to the table name by default (unless the table name is specified in the class using *__tablename__ = "name of table"*. The instance variables of the class (except for the ones that are assigned relationships) correspond to the columns of the table. By migrating all the classes in my *model.py* (which contains all the classes created) to any variance of SQL (SQLite, MySQL, Postgres), the tables are created in the database. Below is the user table.



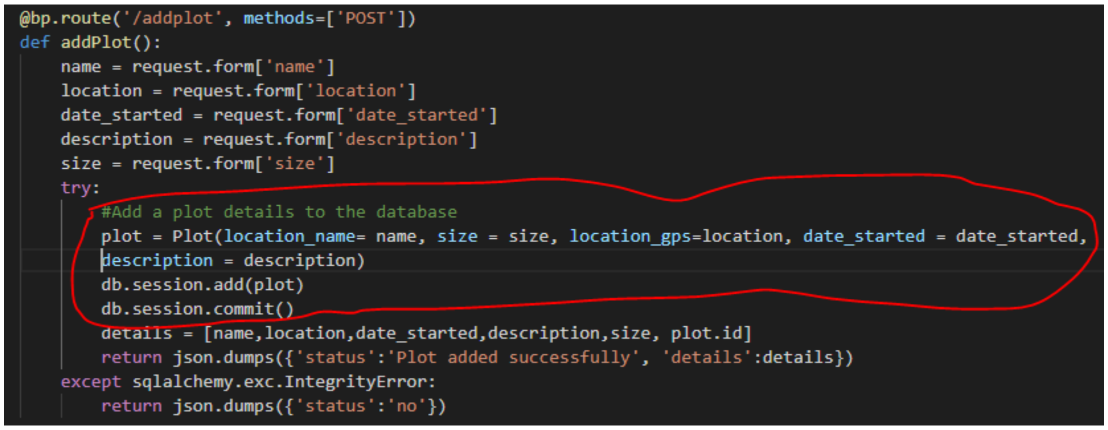
#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	id	int(11)			No	None		AUTO_INCREMENT	Change Drop More
2	name	varchar(64)	latin1_swedish_ci		Yes	NULL			Change Drop More
3	email	varchar(120)	latin1_swedish_ci		Yes	NULL			Change Drop More
4	phone	varchar(120)	latin1_swedish_ci		Yes	NULL			Change Drop More
5	status	varchar(50)	latin1_swedish_ci		Yes	NULL			Change Drop More
6	password_hash	varchar(128)	latin1_swedish_ci		Yes	NULL			Change Drop More
7	security_question	varchar(64)	latin1_swedish_ci		Yes	NULL			Change Drop More
8	security_answer	varchar(64)	latin1_swedish_ci		Yes	NULL			Change Drop More
9	last_message_read_time	datetime			Yes	NULL			Change Drop More

Figure 4.15: User table after model has been migrated to MySQL

To demonstrate the simplicity and the effectiveness of Flask SQLAlchemy for this project, I am going to demonstrate how to add, read, delete, and update data.

4.4.1.1 Adding Data

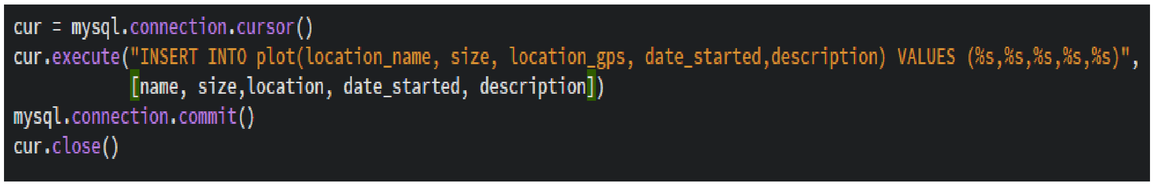
Example: Adding Plot details to the database



```
@bp.route('/addplot', methods=['POST'])
def addPlot():
    name = request.form['name']
    location = request.form['location']
    date_started = request.form['date_started']
    description = request.form['description']
    size = request.form['size']
    try:
        #Add a plot details to the database
        plot = Plot(location_name= name, size = size, location_gps=location, date_started = date_started,
        description = description)
        db.session.add(plot)
        db.session.commit()
        details = [name,location,date_started,description,size, plot.id]
        return json.dumps({'status':'Plot added successfully', 'details':details})
    except sqlalchemy.exc.IntegrityError:
        return json.dumps({'status':'no'})
```

Figure 4.16: Adding data in Flask SQLAlchemy

To add data, the class is imported in the file, and an object of the class is created. The instance variables of the class (which represents the columns of the database) are then assigned to the information that the user enters into the form. The object is then added to the database and once the *db.session.commit()* is invoked, the information is added to the database. This entire process of adding to the database is equivalent to executing the code below in traditional Python SQL operations.



```
cur = mysql.connection.cursor()
cur.execute("INSERT INTO plot(location_name, size, location_gps, date_started,description) VALUES (%s,%s,%s,%s,%s)",
          [name, size,location, date_started, description])
mysql.connection.commit()
cur.close()
```

Figure 4.17: Adding data using traditional SQL queries

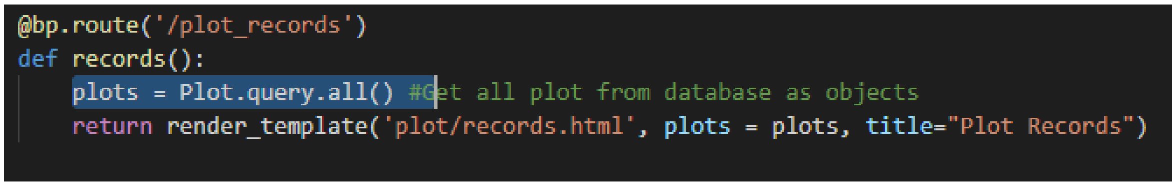
4.4.1.2 Delete Data

Example: Delete Plot from the database (see Figure 4.13 for code snippet)

As shown in Figure 4.13, to delete data from the database, you simply query the object and delete. Once the delete is committed to the database, the data is deleted. In this example, all the details associated with a plot will be deleted. However, Flask SQLAlchemy provides the flexibility to decide whether data associated with a certain object (through foreign key relationships) should be deleted or not.

4.4.1.3 Read Data

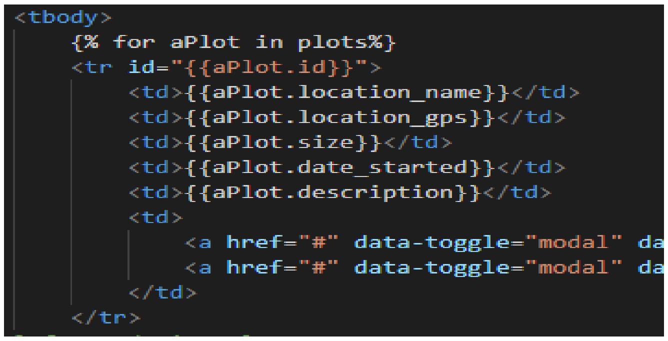
Example: Reading Plot details



```
@bp.route('/plot_records')
def records():
    plots = Plot.query.all() #Get all plot from database as objects
    return render_template('plot/records.html', plots = plots, title="Plot Records")
```

Figure 4.18: Querying data in SQLAlchemy

Perhaps the most powerful and exciting SQLAlchemy operation is reading data from the database. By just carrying out the highlighted code above, a list of all plots recorded is returned as objects. To display these details on a web page, you loop through the list that has been passed through the *render_template* function, and display the details of plots as follows:



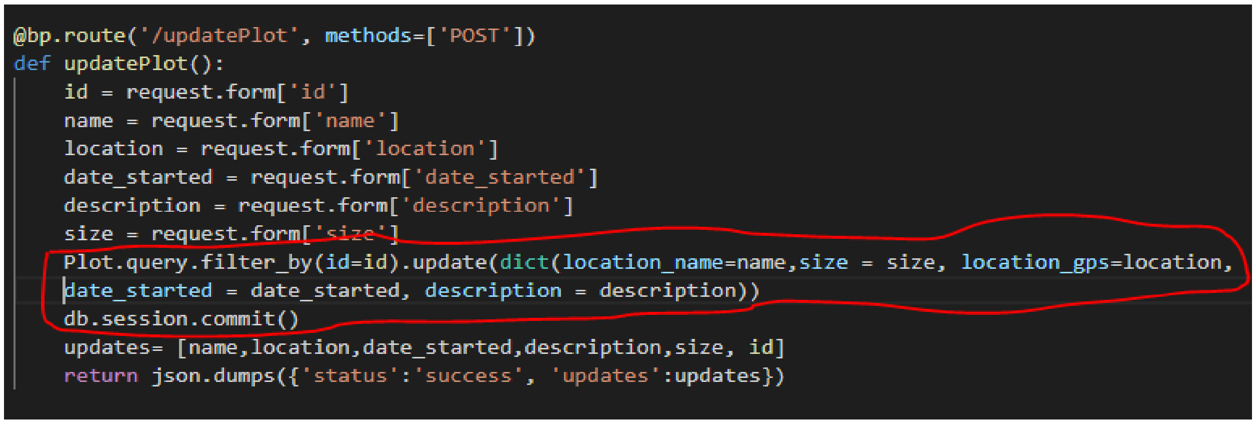
```
<tbody>
    {% for aPlot in plots%}
    <tr id="{{aPlot.id}}">
        <td>{{aPlot.location_name}}</td>
        <td>{{aPlot.location_gps}}</td>
        <td>{{aPlot.size}}</td>
        <td>{{aPlot.date_started}}</td>
        <td>{{aPlot.description}}</td>
        <td>
            <a href="#" data-toggle="modal" da
            <a href="#" data-toggle="modal" da
        </td>
    </tr>
```

Figure 4.19: Display data on HTML page

The result of this is a list of all plots recorded in a table as shown in Figure 4.10 above. But perhaps the most exciting thing is that, say each plot has some crops associated with it, you can do *aPlot.crops* to get a list of all crops associated with a plot (rather than writing a long query to retrieve these details). Similarly, when you want to add associated details in one(many) to many relationships, you can do *aPlot.crops.append(cropObject)*. You are literary doing what you will do when dealing with objects, and once the operation is committed to the database, you get the same result as if the operations were performed using the traditional SQL queries.

4.4.1.4 Update Data

Example: Updating Plot details



```
@bp.route('/updatePlot', methods=['POST'])
def updatePlot():
    id = request.form['id']
    name = request.form['name']
    location = request.form['location']
    date_started = request.form['date_started']
    description = request.form['description']
    size = request.form['size']
    Plot.query.filter_by(id=id).update(dict(location_name=name,size = size, location_gps=location,
    date_started = date_started, description = description))
    db.session.commit()
    updates= [name,location,date_started,description,size, id]
    return json.dumps({'status':'success', 'updates':updates})
```

Figure 4.20: Updating data in SQLAlchemy

To update data, you query the object and then update the variables by assigning new values to them. Once the change is committed to the database, the update is made.

4.4.1.5 Backup

As noted earlier, backup is very necessary for this application. I used a Flask extension called AlchemyDumps [27] to implement thse backup system for the application. This extension allows you to do a backup, generate histories of backups, restore backups, and do an 'autoclean' which clean all backups and keep the most important ones based on certain rules such as 'keep all the backups from the last 7 days' [27].

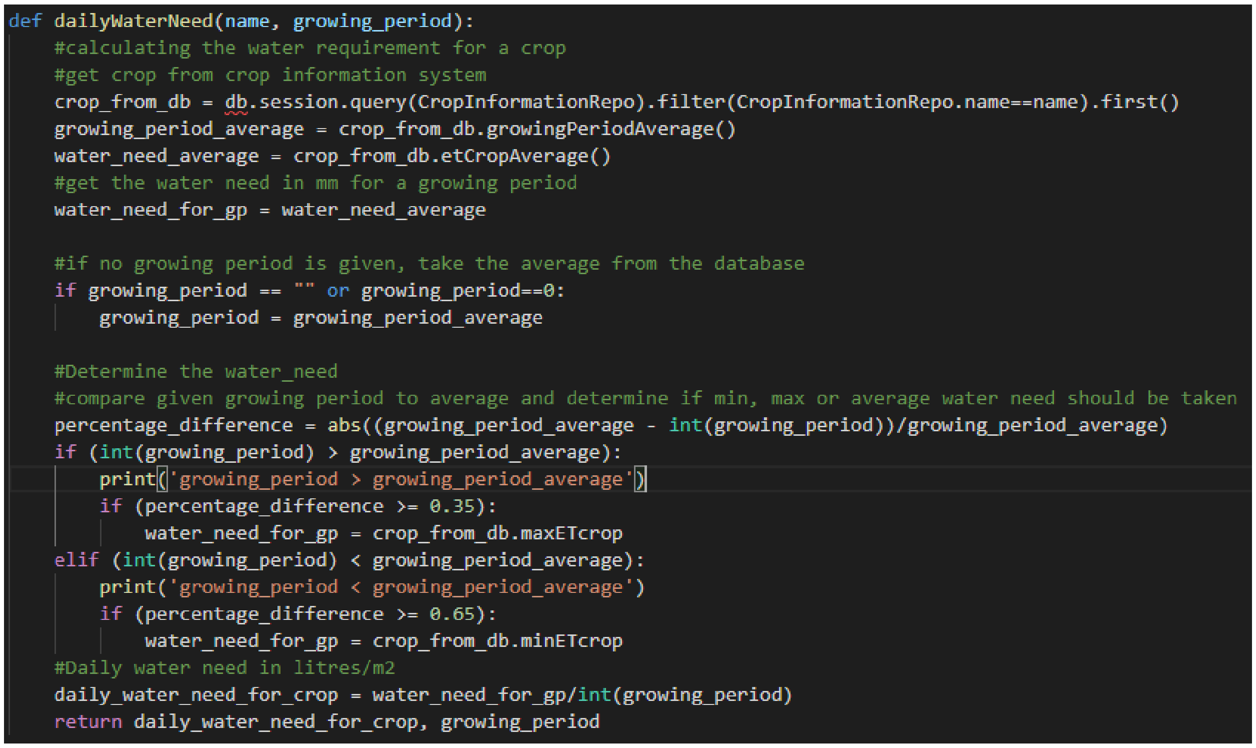
From these demonstrations, it is quite clear that SQLAlchemy simplifies the development process. But aside from simplicity, it offers security and speed. SQLAlchemy prevents SQL injection. "Unless you bypass SQLAlchemy quoting mechanism, SQL injection attacks are impossible" [17]. This is really important for the security of the system. The data that users provide should be protected from malicious attacks. Secondly, because it is object-oriented and the fact that it is specifically designed for Python, performing CRUD operations are faster. These remarkable attributes of SQLAlchemy make it the best choice for this project.

4.5 An exciting feature

One of the exciting functionalities of the system is the water requirement calculator. Water is perhaps the most important resource in farming. Overusing water can cause damage to crops and lead to wastage, and not supplying enough water can cause crops to wilt and die. This is why I considered implementing this functionality.

To implement this functionality, I used a database of common crops with estimates of their growing period and the total amount of water needed for the growing period. This database is provided by the UN's Food and Agriculture Organization (FAO) [20], whose mandate is to improve food security in the world. The water needs of the crops have minimum and maximum values. Also, the growing periods have minimum and maximum values. Here, FAO noted that the range for the growing period for a crop is provided to cater for the different climatic conditions. This software gives users the option to enter the growing period for their locality. This growing period is used to estimate the daily water need for a crop.

When a user selects a crop from this database, the daily water need for the selected plant is calculated using the growing period provided. If no growing period is provided, then the application uses the average growing period of the crop to calculate the daily water need. Below is the code that calculates the daily water need:



```python
def dailyWaterNeed(name, growing_period):
    #calculating the water requirement for a crop
    #get crop from crop information system
    crop_from_db = db.session.query(CropInformationRepo).filter(CropInformationRepo.name==name).first()
    growing_period_average = crop_from_db.growingPeriodAverage()
    water_need_average = crop_from_db.etCropAverage()
    #get the water need in mm for a growing period
    water_need_for_gp = water_need_average

    #if no growing period is given, take the average from the database
    if growing_period == "" or growing_period==0:
        growing_period = growing_period_average

    #Determine the water_need
    #compare given growing period to average and determine if min, max or average water need should be taken
    percentage_difference = abs((growing_period_average - int(growing_period))/growing_period_average)
    if (int(growing_period) > growing_period_average):
        print('growing_period > growing_period_average')
        if (percentage_difference >= 0.35):
            water_need_for_gp = crop_from_db.maxETcrop
    elif (int(growing_period) < growing_period_average):
        print('growing_period < growing_period_average')
        if (percentage_difference >= 0.65):
            water_need_for_gp = crop_from_db.minETcrop
    #Daily water need in litres/m2
    daily_water_need_for_crop = water_need_for_gp/int(growing_period)
    return daily_water_need_for_crop, growing_period
```

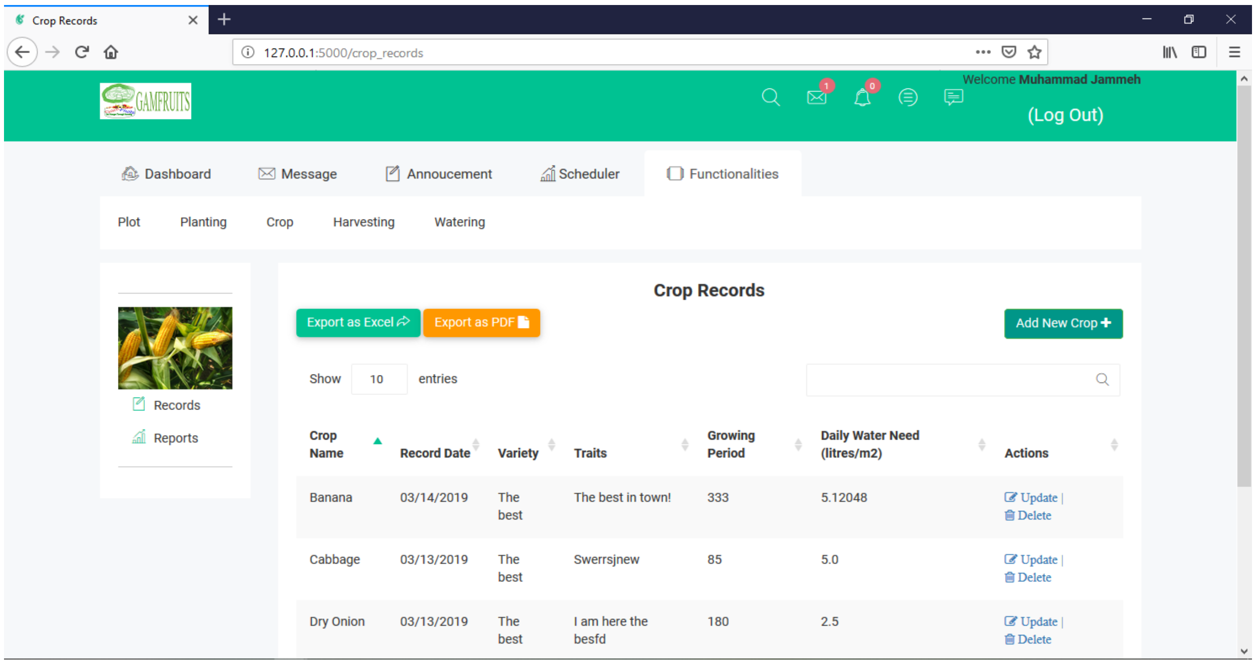
Figure 4.21: Daily water need calculator

The purpose of doing this calculation is to be able to determine the daily water need for a planting. In determining the daily water need for a planting, the size of the plot of land and the crops grown are considered. Typically, the daily water need for a planting is calculated as follows:

$$Daily\ Water\ Need\ for\ Planting = daily\ water\ need\ for\ crop(s)\ in\ litres\ per\ m^2 * plot\ size\ in\ m^2$$

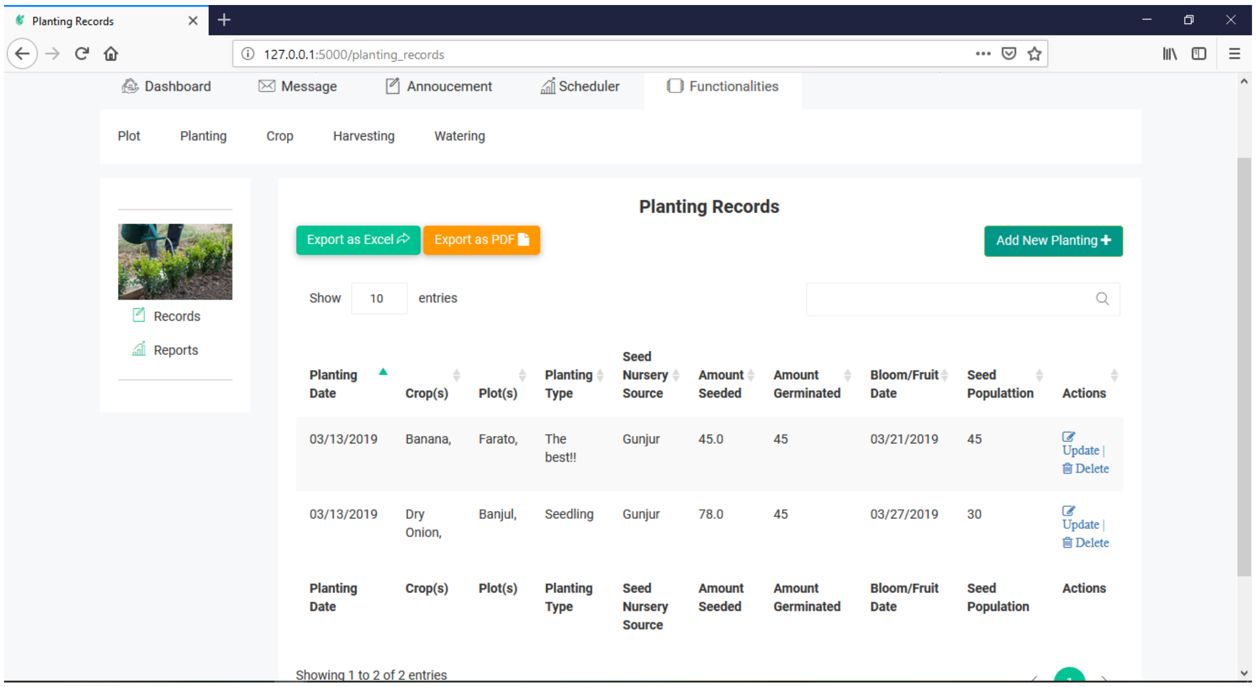
After the daily water need for a planting has been calculated and the user records the watering for a planting in a particular day, if the volume of water recorded is less than the watering need of the planting, periodic notifications are sent to the user. Additionally, the application gives the user the opportunity to record the rainfall (if any) for a particular day. In this case, the effective rainfall (rainfall that is used by the crops, as defined by FAO [9]) is calculated and daily water requirement of the planting is adjusted by subtracting the effective rainfall from the total water need.

4.6 Some Screenshots



Crop Name	Record Date	Variety	Traits	Growing Period	Daily Water Need (litres/m2)	Actions
Banana	03/14/2019	The best	The best in town!	333	5.12048	Update | Delete
Cabbage	03/13/2019	The best	Swerrsjnew	85	5.0	Update | Delete
Dry Onion	03/13/2019	The best	I am here the besfd	180	2.5	Update | Delete

Figure 4.22: Display of recorded crops



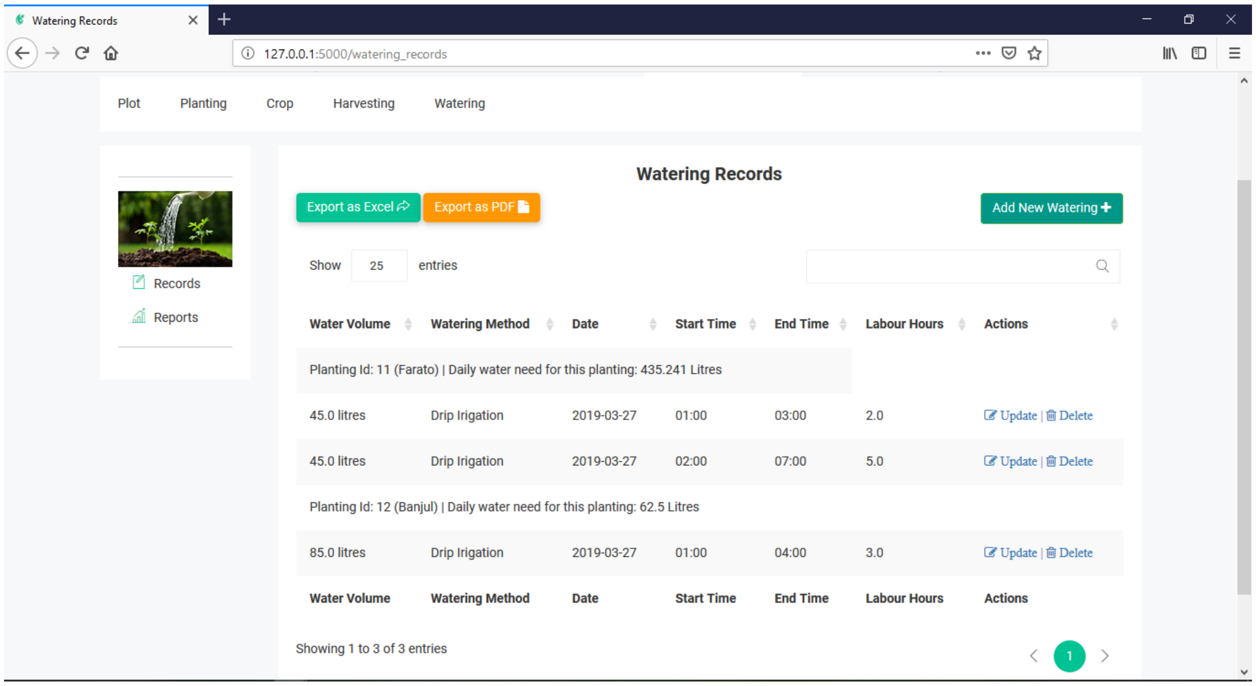
Planting Records

Show 10 entries

Planting Date	Crop(s)	Plot(s)	Planting Type	Seed Nursery Source	Amount Seeded	Amount Germinated	Bloom/Fruit Date	Seed Population	Actions
03/13/2019	Banana,	Farato,	The best!!	Gunjur	45.0	45	03/21/2019	45	Update | Delete
03/13/2019	Dry Onion,	Banjul,	Seedling	Gunjur	78.0	45	03/27/2019	30	Update | Delete
Planting Date	Crop(s)	Plot(s)	Planting Type	Seed Nursery Source	Amount Seeded	Amount Germinated	Bloom/Fruit Date	Seed Population	Actions

Showing 1 to 2 of 2 entries

Figure 4.23: Display for planting records



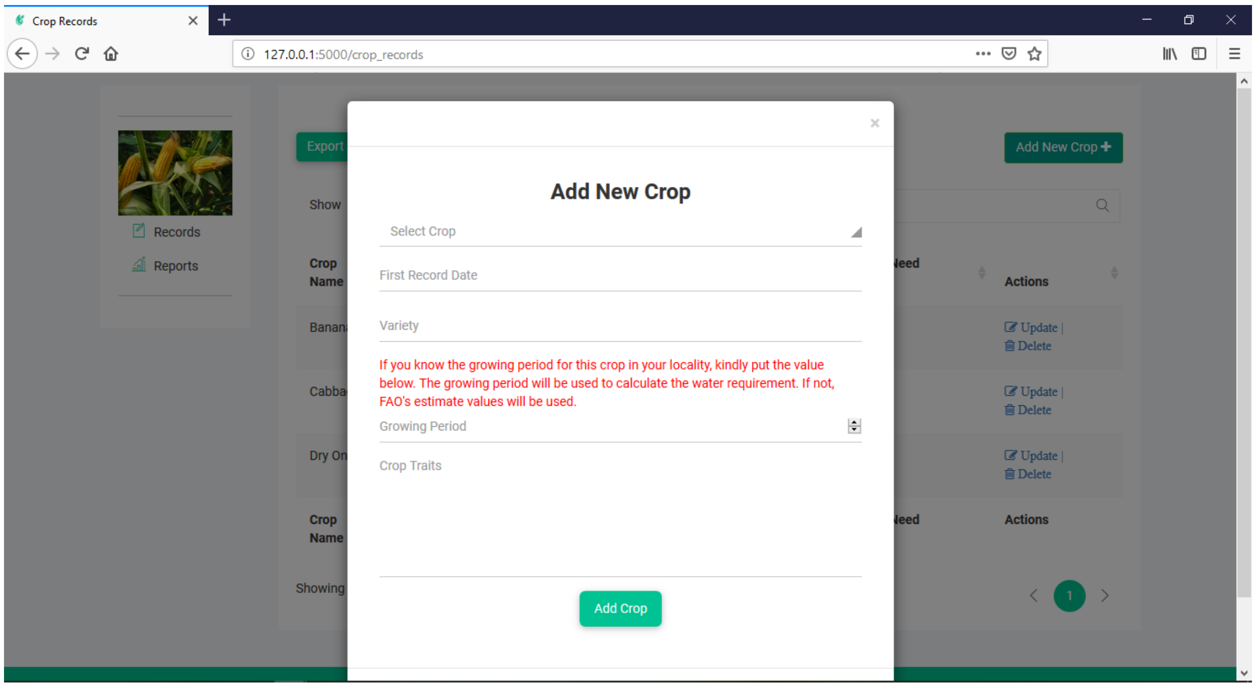
Watering Records

Show 25 entries

Water Volume	Watering Method	Date	Start Time	End Time	Labour Hours	Actions
Planting Id: 11 (Farato) | Daily water need for this planting: 435.241 Litres						
45.0 litres	Drip Irigation	2019-03-27	01:00	03:00	2.0	Update | Delete
45.0 litres	Drip Irigation	2019-03-27	02:00	07:00	5.0	Update | Delete
Planting Id: 12 (Banjul) | Daily water need for this planting: 62.5 Litres						
85.0 litres	Drip Irigation	2019-03-27	01:00	04:00	3.0	Update | Delete
Water Volume	Watering Method	Date	Start Time	End Time	Labour Hours	Actions

Showing 1 to 3 of 3 entries

Figure 4.24: Watering Records



Crop Records

127.0.0.1:5000/crop_records

Add New Crop

Select Crop

First Record Date

Variety

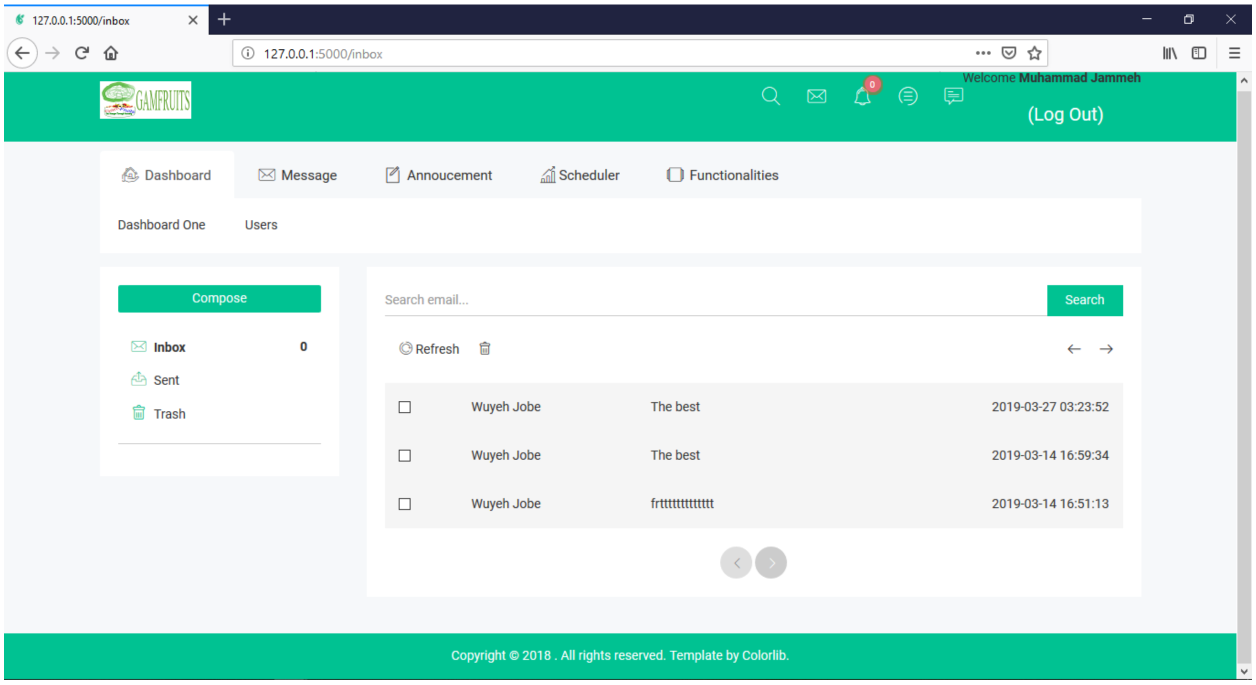
If you know the growing period for this crop in your locality, kindly put the value below. The growing period will be used to calculate the water requirement. If not, FAO's estimate values will be used.

Growing Period

Crop Traits

Add Crop

Figure 4.25: Adding a new crop. After adding crop, daily water need is calculated



127.0.0.1:5000/inbox

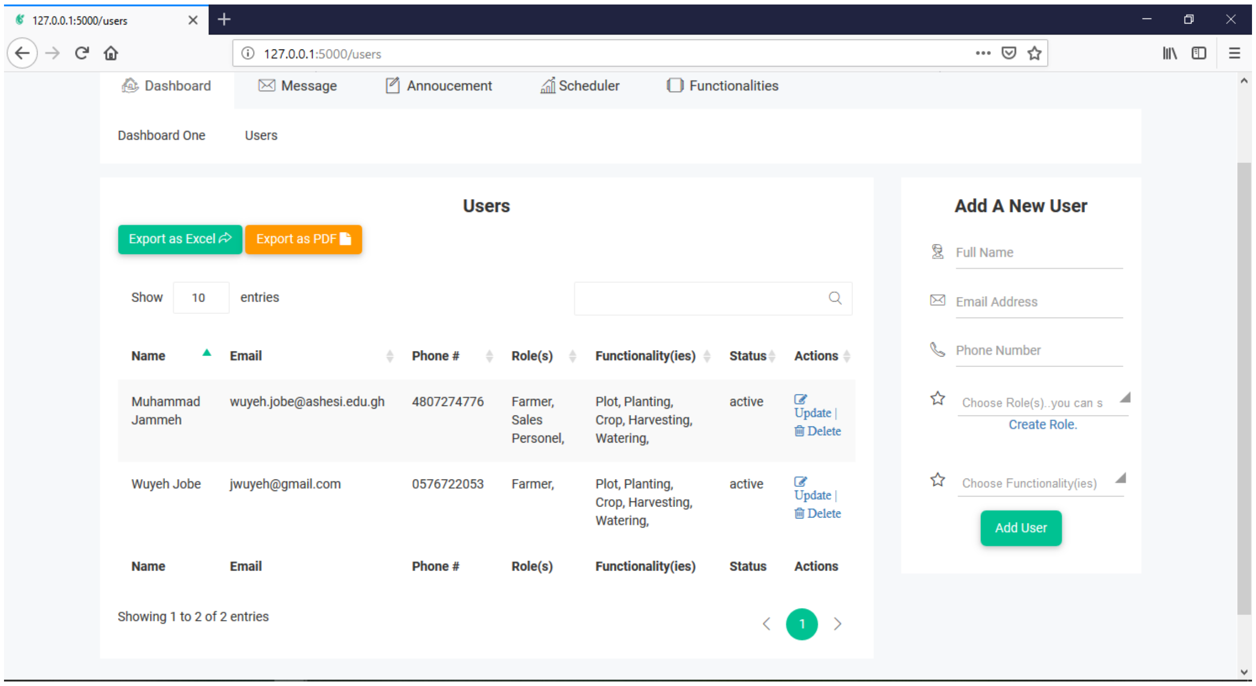
Welcome **Muhammad Jammeh**

(Log Out)

☐	Wuyeh Jobe	The best	2019-03-27 03:23:52
☐	Wuyeh Jobe	The best	2019-03-14 16:59:34
☐	Wuyeh Jobe	frtttttttttttt	2019-03-14 16:51:13

Copyright © 2018 . All rights reserved. Template by Colorlib.

Figure 4.26: Messaging system



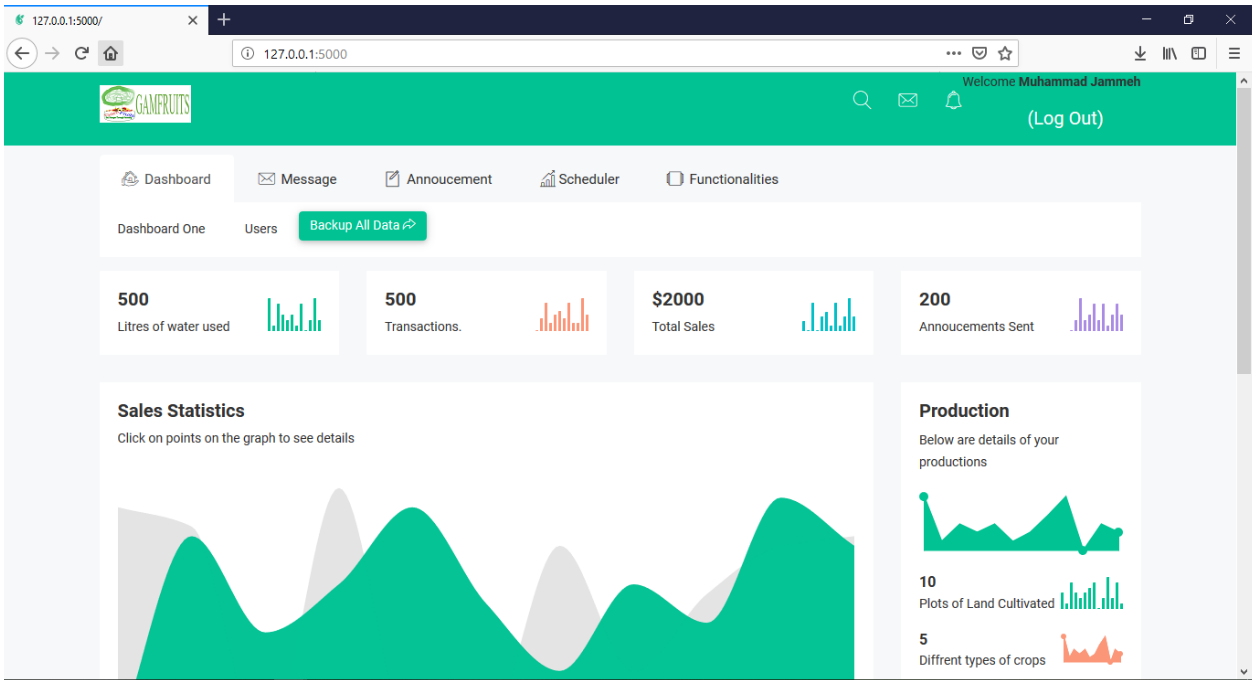
Users

Name	Email	Phone #	Role(s)	Functionality(ies)	Status	Actions
Muhammad Jammeh	wuyeh.jobe@ashesi.edu.gh	4807274776	Farmer, Sales Personel,	Plot, Planting, Crop, Harvesting, Watering,	active	Update | Delete
Wuyeh Jobe	jwuyeh@gmail.com	0576722053	Farmer,	Plot, Planting, Crop, Harvesting, Watering,	active	Update | Delete
Name	Email	Phone #	Role(s)	Functionality(ies)	Status	Actions

Showing 1 to 2 of 2 entries

Add A New User

Figure 4.27 Creating Users and the functionalities they can access



500 Litres of water used

500 Transactions.

$2000 Total Sales

200 Annoucements Sent

Sales Statistics

Click on points on the graph to see details

Production

Below are details of your productions

10 Plots of Land Cultivated

5 Diffrent types of crops

Figure 4.28: Dashboard

Chapter 5 - Testing and Results

Undoubtedly, testing is a crucial aspect of software development. It helps developers to fish out bugs that may hamper the usability and efficiency of the system. For this system, testing was carried out in two main stages: development testing and user testing.

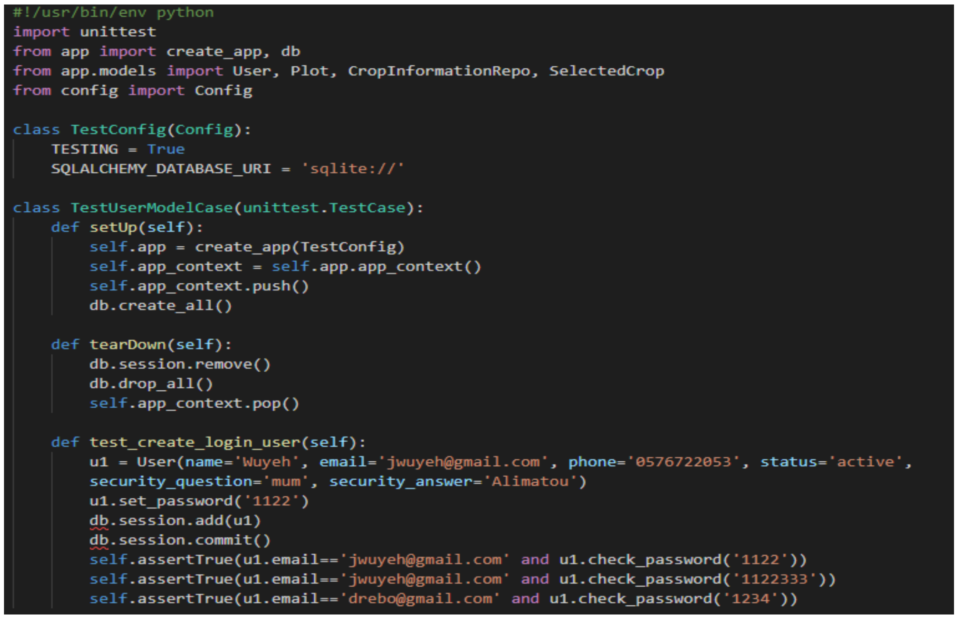
5.1 Development Testing

The three main development testing used for developing this system are unit testing, component testing, and system testing.

5.1.1 Unit Testing

We provide unit testing for a subset of all identifiable test cases in this project. This is mainly due to time constraints. Here, crucial units that form the foundation of each subsystem were tested. For instance, in the authentication subsystem, the units for registering a user, sending an email to a newly registered user, and ensuring that a registered user can log in were tested. In the planting and harvesting subsystem, the units for adding a plot and crop, and the unit for calculating the water need for a crop were tested. And so on.

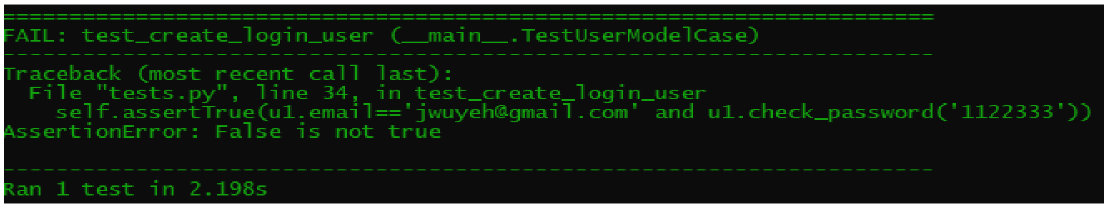
To conduct the tests, a python test script was created. In the script, the application was connected to a dummy database and the unit tests were carried out using this database. Below is the format of the *tests.py* file that shows how to do tests for the units.



```python
#!/usr/bin/env python
import unittest
from app import create_app, db
from app.models import User, Plot, CropInformationRepo, SelectedCrop
from config import Config

class TestConfig(Config):
    TESTING = True
    SQLALCHEMY_DATABASE_URI = 'sqlite://'

class TestUserModelCase(unittest.TestCase):
    def setUp(self):
        self.app = create_app(TestConfig)
        self.app_context = self.app.app_context()
        self.app_context.push()
        db.create_all()

    def tearDown(self):
        db.session.remove()
        db.drop_all()
        self.app_context.pop()

    def test_create_login_user(self):
        u1 = User(name='Wuyeh', email='jwuyeh@gmail.com', phone='0576722053', status='active',
        security_question='mum', security_answer='Alimatou')
        u1.set_password('1122')
        db.session.add(u1)
        db.session.commit()
        self.assertTrue(u1.email=='jwuyeh@gmail.com' and u1.check_password('1122'))
        self.assertTrue(u1.email=='jwuyeh@gmail.com' and u1.check_password('1122333'))
        self.assertTrue(u1.email=='drebo@gmail.com' and u1.check_password('1234'))
```

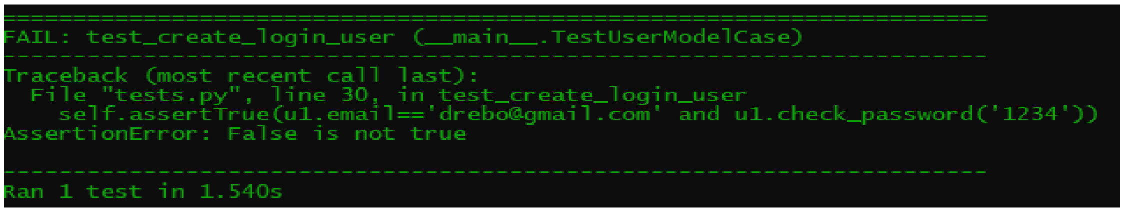
Figure 5.1: Unit testing in Python

Observe that in the file, I have a function called *test_create_login_user* that tests if a user is created and that a user with the correct credentials can log in. When this test is run, the last two lines will fail because the correct password was not used in the first one and the email used in the second does not match that of the registered user. Below are the results of the test:



```
======================================================================
FAIL: test_create_login_user (__main__.TestUserModelCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "tests.py", line 34, in test_create_login_user
    self.assertTrue(u1.email=='jwuyeh@gmail.com' and u1.check_password('1122333'))
AssertionError: False is not true

----------------------------------------------------------------------
Ran 1 test in 2.198s
```

Figure 5.2: Failed test because of a wrong password



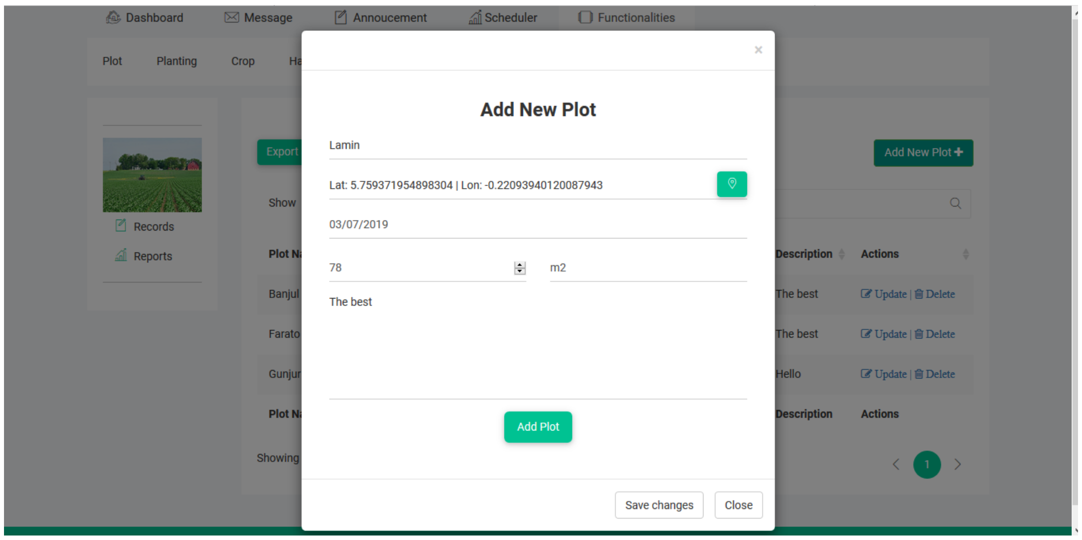
```
======================================================================
FAIL: test_create_login_user (__main__.TestUserModelCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "tests.py", line 30, in test_create_login_user
    self.assertTrue(u1.email=='drebo@gmail.com' and u1.check_password('1234'))
AssertionError: False is not true

----------------------------------------------------------------------
Ran 1 test in 1.540s
```

Figure 5.3: Failed test because of a wrong email

The units for other subsystems are tested in a similar manner.

5.1.2 Component Testing

After the units of the subsystems have been successfully tested, they were put together to form components. For instance, after testing add, delete and update for Plot Records, the units were integrated with the Plot records HTML template to enable the user to interact with the system.



Add New Plot

Lamin

Lat: 5.759371954898304 | Lon: -0.22093940120087943

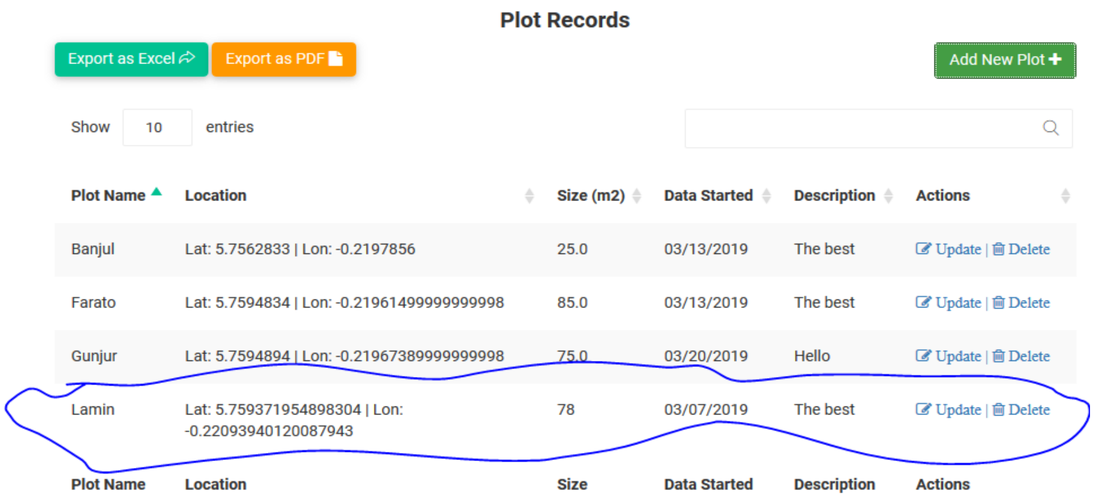
03/07/2019

78 m2

The best

Figure 5.4: A component test for adding plot

When the user adds a Plot from the interface, the details of the plot are recorded in the database and displayed to the user in the Plot Records table as shown below:



Plot Records

Plot Name	Location	Size (m2)	Data Started	Description	Actions
Banjul	Lat: 5.7562833 | Lon: -0.2197856	25.0	03/13/2019	The best	Update | Delete
Farato	Lat: 5.7594834 | Lon: -0.21961499999999998	85.0	03/13/2019	The best	Update | Delete
Gunjur	Lat: 5.7594894 | Lon: -0.21967389999999998	75.0	03/20/2019	Hello	Update | Delete
Lamin	Lat: 5.759371954898304 | Lon: -0.22093940120087943	78	03/07/2019	The best	Update | Delete
Plot Name	Location	Size	Data Started	Description	Actions

Figure 5.5: Result of adding Plot

The same thing happens for update and delete. For an example of how the delete works, see the explanation given for figure 4.10 above. All the components of the subsystems were tested in a similar manner.

5.1.3 System Testing

For system testing, all the different components of the system were tested at a go. For instance, to demonstrate how the software helps in tracking the planting and harvesting history of a farm, a simulation of a farming season was done, and everything from planting to harvesting was recorded, and how the software helped in calculating the water requirement needs of the farm. See the video[6] of the system in action!

5.2 User Testing

For user testing, the system was hosted online and an agripreneur went through it to identify if bugs exist in the system and if there are areas that need to be improved. One of the most important feedback received is that 'datetime' input field does not work in Firefox. During development, I was mostly using chrome for testing and this testing was really useful in identifying this loophole. Other feedback included issues to deal with interface design. The agripreneur suggested having a theme selector that will allow user to change the theme of the application to suit their needs. He also stated that he want to be able to create dynamic data fields to have the opportunity to add data that is necessary for his own use. Overall, the agripreneur was impressed by usefulness of the application and cannot wait to deploy it for his own use.

[6] https://drive.google.com/file/d/1ntKN6UjVVvu5K_gXQO3x2C3DrRbiY9r9/view?usp=sharing

Chapter 6 - Conclusions, Recommendations, and Future work

The purpose of this project is to build a resource planning software for youth Agripreneurs on the African continent. The functionalities implemented are: water and fertilizer requirement calculators; a tool to track the planting, watering, and harvesting history of a farm; a tool to manage the sales of products; an announcement broadcaster; a payroll manager; a scheduler for agricultural operations which get converted to work done list; a messaging system; a data sharing tool; a backup system; and charts for reporting purposes.

6.1 Challenges

Of course, this project pushed me to my limit. It challenged me to think critically and explore technologies and techniques to create a tool that will be of benefit to society. From the planning to the execution stage, it compelled me to reflect on the knowledge I have accumulated over the years from the courses I took at Ashesi.

The greatest challenge I was faced with was evaluating the technologies to use for the system. I spent about a month trying to evaluate the technologies I should use. I read documentation and tutorials, watched videos, and spoke with advisors. Because for me, this project was not just about completing my undergraduate studies. It was not just about showing my programming prowess. It was not about building a system in the best way I know how. This project was about building a system that will stand the test of time. It was about designing a solution that takes into consideration technological trends and the impact they have on the software system we create.

Thus, I was very intentional about the choice of Python Flask as the main programming language for the backend. Because I want my system to make use of machine learning in the future, and since Python is the main language for machine learning, Python Flask gives me the opportunity to integrate machine learning models to make the system more impactful in the future. Making that decision was scary in the beginning because I had very little knowledge of the web framework compared to my knowledge in PHP. But I made sure that do not deter me from using the language that I think would be best for the project. Through online tutorials and practice, I made sure that I did not only use the language but also ensure that I follow best practices.

My second biggest challenge was to combine different technologies to get the application to work as efficiently as possible. Although the description of the development process may look simple, it is not. There is a lot of work that went into combining all these technologies. For instance, using Ajax to make request to the server comes with the extra burden of ensuring that the HTML elements can be manipulated even though the page is not reloaded. For areas that require generating dynamic content based on user interaction and the need to fetch data from the database, the challenges encountered are numerous. For instance, when recording harvesting details, the yield to be recorded is depended on the crops that are associated with each planting. So once, the user selects a planting to be harvested, all the crops associated with a planting (a planting is also associated with plots) are retrieved, and input fields are generated for each crop to record the yield value and the unit of measurement. All these are done without reloading the page. Doing things like this become easier when a page is reloaded, but for the sake of usability, you have to find a way around to ensure that the user enjoys using the functionality (i.e. avoiding page reload).

My final biggest challenge was implementing the backup system. The documentation on how to use it was scanty. The most difficult part though was creating an interface for the backup system. The backup system was designed to run using command line prompts and so I had to figure out a way to interact with it using a button click. Although I figured it out, it took a lot of time and research.

6.2 Future Work

Indeed, there is a lot of work that goes into building a resource planning software. Thus, for this system, there are a lot more features to implement.

Firstly, a livestock management subsystem will be implemented for youth Agripreneurs who engage in animal husbandry.

Secondly, machine learning models will be built to make more accurate estimates and predictions about the water and fertiliser requirements of the crops.

Thirdly, an accounting subsystem will be built to simplify the financial accounting system for the youth agripreneurs.

Fourthly, functions will be created to give way for IoT devices to interact with the system.

Finally, a website will be created to provide a step by step approach to users on how to install and use the system.

Additionally, the human computer interaction (HCI) aspect of the system should be improved to allow agripreneurs to record and retrieve data swiftly.

Finally, the system should integrate a weather system for use in other functionalities such as water requirement calculator.

6.3 Recommendations

In the meantime, before I build the livestock management and accounting subsystem, I will recommend that youth Agripreneurs who engage in animal husbandry should check out a livestock management software called Tambero.com [18]. Although this may be difficult to use and not be a good fit for their needs, it has a lot of features to help them get started. Once the livestock management subsystem is built, they can migrate their data.

The accounting software I will recommend is called Akaunting [19]. It is a generic accounting software for almost all types of business. The Agripreneurs can explore it to see if it will be useful for their activities.

References

[1] World population projected to reach 9.6 billion by 2050 | UN DESA Department of Economic and Social Affairs. (June 2013). Retrieved September 26, 2018 from http://www.un.org/en/development/desa/news/population/un-report-world-population-projected-to-reach-9-6-billion-by-2050.html

[2] Tom Huston. 2017. How do we feed the planet in 2050? (September 2017). Retrieved September 26, 2018 from https://www.theguardian.com/preparing-for-9-billion/2017/sep/13/population-feed-planet-2050-cold-chain-environment

[3] Erick Oduor, Peninah Waweru, Jonathan Lenchner, and Carman Neustaedter. 2018. Practices and Technology Needs of a Network of Farmers in Tharaka Nithi, Kenya. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (CHI '18). ACM, New York, NY, USA, Paper 39, 11 pages. DOI: https://doi.org/10.1145/3173574.3173613

[4] Mohamed El Mohadab, Belaid Boui Khalene, and Said Safi. 2017. Enterprise resource planning: Introductory overview. *2017 International Conference on Electrical and Information Technologies (ICEIT) (2017).* DOI: http://dx.doi.org/10.1109/eitech.2017.8255306

[5] Todor Stoilov and Krasimira Stoilova. 2008. Functional analysis of enterprise resource planning systems. In *Proceedings of the 9th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing* (CompSysTech '08), Boris Rachev and Anger Smrikarov (Eds.). ACM, New York, NY, USA, Article 43. DOI: http://dx.doi.org/10.1145/1500879.1500927

[6] Donald Byamugisha. The Agripreneur: A new breed of young entrepreneurs combining their love of Farming and Agriculture with an acquired professional Business Approach. Retrieved

October 10, 2018 from https://ypard.net/testimonials/agripreneur-new-breed-young-entrepreneurs-combining-their-love-farming-and-agriculture-

[7] Francis N.T Yinbill. 2017. An online system that connect farmers to buyers. (April 2017). Bachelor's applied project. Ashesi University. Retrieved October 10, 2018 from http://hdl.handle.net/20.500.11988/300

[8] Ndubuisi Ekekwe. 2017. How Digital Technology Is Changing Farming In Africa (May 2017). Retrieved November 20, 2018 from https://hbr.org/2017/05/how-digital-technology-is-changing-farming-in-africa

[9] FAO. Chapter 4: Irrigation Water Need. Retrieved April 1, 2019 from http://www.fao.org/3/s2022e/s2022e08.htm

[10] Flask. Flask Web Development, one drop at a time. Retrieved March 16, 2019 from http://flask.pocoo.org/

[11] Forward. Retrieved March 16, 2019 from http://flask.pocoo.org/docs/1.0/foreword/#what-does-micro-mean

[12] Miguel Grinberg. 2018. The Flask Mega-Tutorial Part XV: A Better Application Structure (March 2018). Retrieved March 17, 2019 from https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-xv-a-better-application-structure

[13] Notika – Free Bootstrap admin dashboard template. Retrieved March 17, 2019 from https://themewagon.com/themes/free-bootstrap-admin-dashboard-template-notika/

[14] Free Bootstrap admin dashboard. Retrieved March 17, 2019 from https://github.com/puikinsh/notika

[15] What Is Ajax Programming – Explained. 2018. Retrieved March 17, 2019 from https://www.keycdn.com/support/ajax-programming

[16] Flask – SQLAlchemy. Retrieved March 18, 2019 from https://www.tutorialspoint.com/flask/flask_sqlalchemy.htm

[17] A step-by-step SQLAlchemy Tutorial. Retrieved March 20, 2019 from http://www.rmunn.com/sqlalchemy-tutorial/tutorial.html

[18] Tambero.com. Retrieved April 1, 2019 from https://www.tambero.com/en

[19] Free Accounting Software. Retrieved April 1, 2019 from https://akaunting.com/

[20] FAO. Chapter 3: Crop Water Needs. Retrieved April 1, 2019 from www.fao.org/3/s2022e/s2022e07.htm

[21] UjuziKilimo. Technology + Data for Quality Modern Farming. Retrieved from https://www.ujuzikilimo.com/

[22] Farmerline. Market-driven solutions empowering the entire agric value chain. Retrieved from https://farmerline.co/

[23] Agrospaces. Supporting Farmers across Africa. Retrieved from https://agrospaces.com/

[24] FarmERP. Retrieved from https://www.farmerp.com/

[25] Arnaud Gallet. 2018. New technologies key to reforming Africa’s agriculture sector (September 2018). Retrieved April 23, 2019 from https://africatimes.com/2018/09/13/new-technologies-key-to-reforming-africas-agriculture-sector/

[26] African Union and NEPAD. Drones on the horizon: Transforming Africa’s Agriculture. Retrieved April 23, 2019 from http://www.nepad.org/publication/drones-horizon-transforming-africas-agriculture

[27] Eduardo Cuducos. (2015). SQLAlchemy backup/dump tool for flask. Retrieved April 23, 2019 from https://github.com/cuducos/alchemydumps

Appendix

Application Folder Tree

```
📦app
 ┣ 🗀auth
 ┃ ┣ 🗎emails.py
 ┃ ┣ 🗎forms.py
 ┃ ┣ 🗎routes.py
 ┃ └ 🗎__init__.py
 ┣ 🗀crop
 ┃ ┣ 🗎routes.py
 ┃ └ 🗎__init__.py
 ┣ 🗀harvesting
 ┃ ┣ 🗎routes.py
 ┃ └ 🗎__init__.py
 ┣ 🗀main
 ┃ ┣ 🗎routes.py
 ┃ └ 🗎__init__.py
 ┣ 🗀messaging
 ┃ ┣ 🗎routes.py
 ┃ └ 🗎__init__.py
 ┣ 🗀planting
 ┃ ┣ 🗎routes.py
 ┃ └ 🗎__init__.py
 ┣ 🗀plot
 ┃ ┣ 🗎routes.py
 ┃ └ 🗎__init__.py
 ┣ 🗀static
 ┃ ┣ 🗀css
 ┃ ┣ 🗀fonts
 ┃ ┣ 🗀images
 ┃ ┣ 🗀js
 ┣ 🗀templates
 ┃ ┣ 🗀auth
 ┃ ┃ ┣ 🗎login.html
 ┃ ┃ ┣
🗎password_request_form.html
 ┃ ┃ ┣ 🗎register.html
 ┃ ┃ └
🗎request_password_reset.html
s
```

```
 ┃ ┣ 🗀crop
 ┃ ┃ ┣ 🗎records.html
 ┃ ┃ └ 🗎reports.html
 ┃ ┣ 🗀email
 ┃ ┃ ┣ 🗎account_setup.html
 ┃ ┃ ┣ 🗎account_setup.txt
 ┃ ┃ ┣ 🗎reset_password.html
 ┃ ┃ └ 🗎reset_password.txt
 ┃ ┣ 🗀harvesting
 ┃ ┃ ┣ 🗎records.html
 ┃ ┃ └ 🗎reports.html
 ┃ ┣ 🗀main
 ┃ ┃ ┣ 🗎first_setup.html
 ┃ ┃ ┣ 🗎index.html
 ┃ ┃ ┣ 🗎info.html
 ┃ ┃ └ 🗎users.html
 ┃ ┣ 🗀messaging
 ┃ ┃ ┣ 🗎compose.html
 ┃ ┃ ┣ 🗎inbox.html
 ┃ ┃ └ 🗎view.html
 ┃ ┣ 🗀planting
 ┃ ┃ ┣ 🗎records.html
 ┃ ┃ └ 🗎reports.html
 ┃ ┣ 🗀plot
 ┃ ┃ ┣ 🗎records.html
 ┃ ┃ └ 🗎reports.html
 ┃ ┣ 🗀watering
 ┃ ┃ ┣ 🗎records.html
 ┃ ┃ └ 🗎reports.html
 ┃ ┣ 🗎layout.html
 ┃ └ 🗎_messages.html
 ┣ 🗀watering
 ┃ ┣ 🗎routes.py
 ┃ └ 🗎__init__.py
 ┣ 🗎email.py
 ┣ 🗎models.py
 └ 🗎__init__.py
```