FarmPulse

Project Engineering

Year 4

Paddy Downey

Bachelor of Engineering (Honours) in Software and Electronic Engineering

Atlantic Technological University

2024/2025

Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Honours) in Software and Electronic Engineering Atlantic Technological University Galway.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

Paddy Downey

Acknowledgements

I would like to take this chance to acknowledge all my Lecturers and academic staff here in ATU Galway for all the support, learning and guidance they have provided me over the last 4 years. I would like to thank Brian O Shea for being my project supervisor and providing me with the right guidance throughout my Final Year Project. I would also like to thank all my Family and friends for the general support over the years.

Lastly, I would like to give a warm thank you to all my fellow classmates. I have made many friends here for life throughout my four years and wish you all the best in your professional life.

Table of Contents

1 Table of Figures

2 Summary

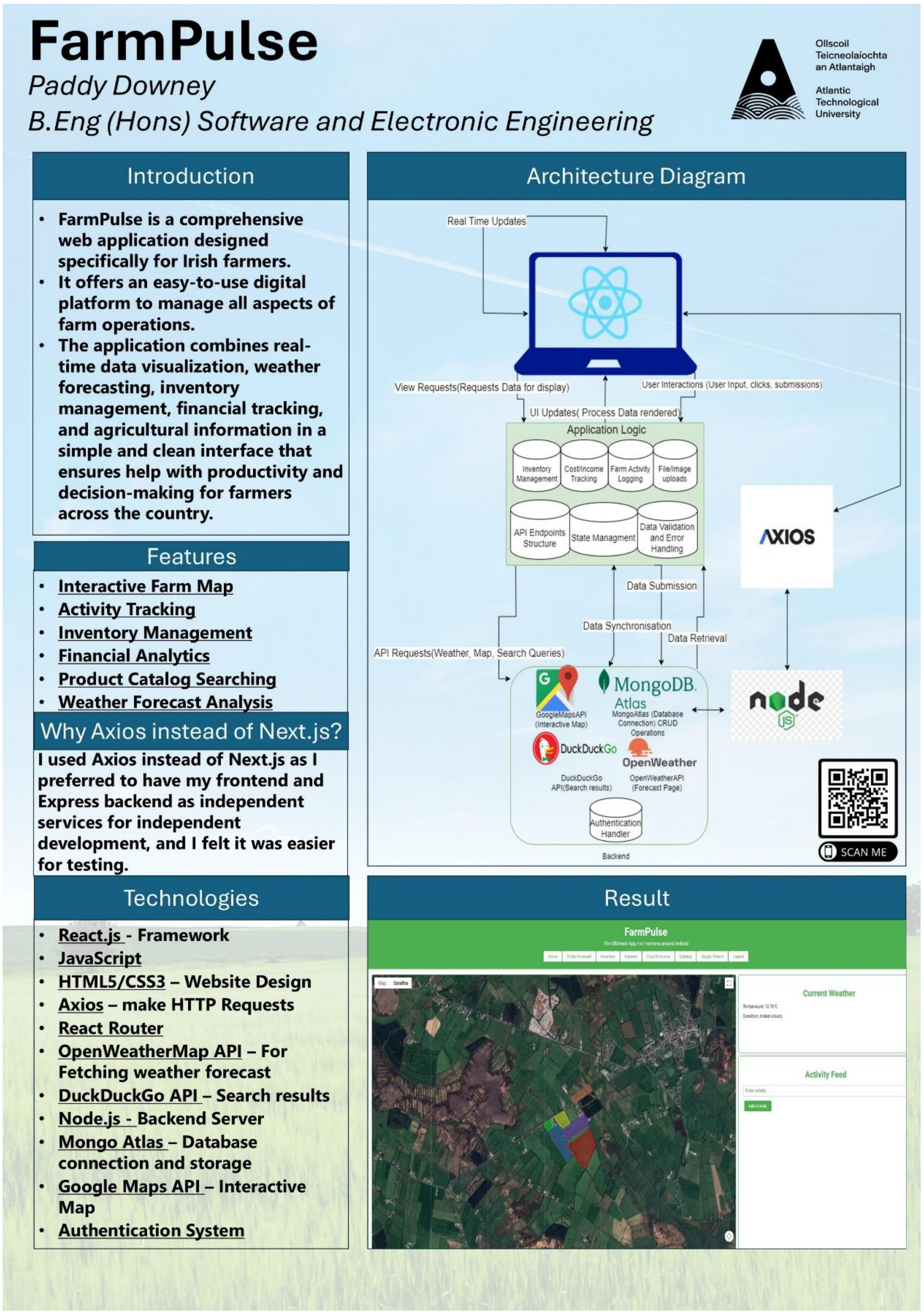
Farming has taken off over the last few years, becoming more and more popular, with overall bigger farms, bigger machinery, more stock, and more money involved than ever before! The goal of this project is to provide a one stop application for Farmers across the island of Ireland, to keep track of their farm stock, activities, income, expenditure, and giving them comprehensive weather forecasts, making life that little bit easier for them through the busy farming seasons. The projects scope focused on Geographic Coverage of farms across Ireland, while supporting Field Management, Weather monitoring and ensuring Data storage to let the farmer keep track of their stock and income/expenditure.

This application is built using React for the frontend and node.js/Express for the backend, with Axios for the API calls and MongoDB serves as the database. The architecture follows a single-page-application (SPA) design with component-based structure. Authentication is implemented using context-based state management, and the app also features a user-friendly light/dark theme support.

The main methods were – RESTful API endpoints for CRUD operations, Real time weather integration using OpenWeatherMap API, Google maps integration for field visualization, Web catalog search results using Serp API, Image capture and storage functionality, State management using React Context and Responsive Data visualization using Recharts.

I accomplished a lot during this project. I successfully implemented a user-friendly interface for farm management while integrating multiple third-party APIs. I also created a scalable database structure for farm data. I implemented real time weather monitoring and forecasting while also developing a comprehensive activity tracking system. The project successfully delivers a modern integrated solution for farm management that combines essential tools and features all in one platform.

3 Poster



FarmPulse

Paddy Downey

B.Eng (Hons) Software and Electronic Engineering

Ollscoil Teicneolaíochta an Atlantaigh
Atlantic Technological University

Introduction

- FarmPulse is a comprehensive web application designed specifically for Irish farmers.
- It offers an easy-to-use digital platform to manage all aspects of farm operations.
- The application combines real-time data visualization, weather forecasting, inventory management, financial tracking, and agricultural information in a simple and clean interface that ensures help with productivity and decision-making for farmers across the country.

Features

- Interactive Farm Map
- Activity Tracking
- Inventory Management
- Financial Analytics
- Product Catalog Searching
- Weather Forecast Analysis

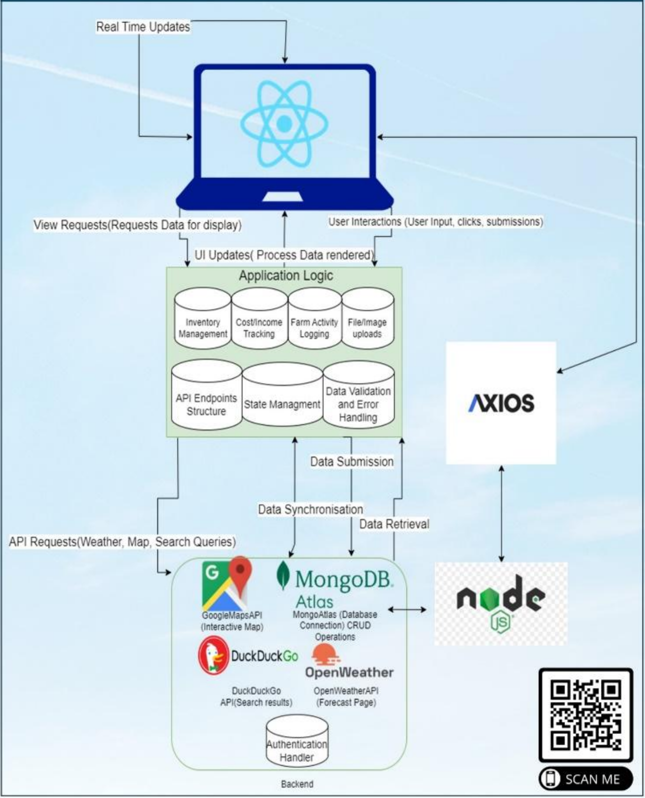
Why Axios instead of Next.js?

I used Axios instead of Next.js as I preferred to have my frontend and Express backend as independent services for independent development, and I felt it was easier for testing.

Technologies

- React.js - Framework
- JavaScript
- HTML5/CSS3 – Website Design
- Axios – make HTTP Requests
- React Router
- OpenWeatherMap API – For Fetching weather forecast
- DuckDuckGo API – Search results
- Node.js - Backend Server
- Mongo Atlas – Database connection and storage
- Google Maps API – Interactive Map
- Authentication System

Architecture Diagram


Real Time Updates
View Requests(Requests Data for display)
User Interactions (User Input, clicks, submissions)
UI Updates(Process Data rendered)
Application Logic
Inventory Management
Cost/Income Tracking
Farm Activity Logging
File/Image uploads
API Endpoints Structure
State Managment
Data Validation and Error Handling
AXIOS
Data Submission
Data Synchronisation
Data Retrieval
API Requests(Weather, Map, Search Queries)
GoogleMapsAPI (Interactive Map)
MongoDB Atlas
MongoAtlas (Database Connection) CRUD Operations
node
DuckDuckGo
OpenWeather
DuckDuckGo API(Search results)
OpenWeatherAPI (Forecast Page)
Authentication Handler
Backend
SCAN ME

Result

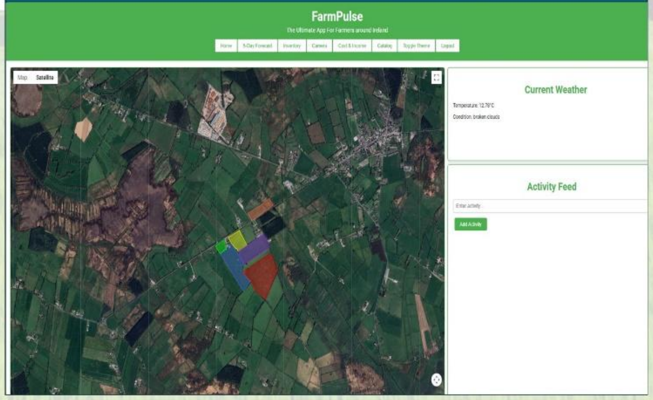


Figure 1: Project Poster

4 Introduction

Since I was younger, I have always had a huge interest in Farming. This stems from my ancestors, all being risen on a farm, and being passed down generations. My first memory of farming was with my Grandad when I was 4 years old, containing simple jobs like feeding the sheep and pigs that we used to own. Nowadays, I am really involved in my family farm, taking up some time in the evenings and weekends to help my dad around the farm, and going to the mart to sell our stock. I also always had a deep interest in electronics since I was younger, always being curious in them and being the go-to in my family for all problems with their phones, laptops, televisions and tablets. This is why I came up with the idea of Farm Pulse. My personal connection has driven me to enhance the development of practical farming features that addresses real-world agricultural management challenges.

I aimed to make a user-friendly app, combining the 2 loves of my life, farming and electronics. I thoroughly enjoyed coming up with the proposal for my idea. Farming is becoming bigger in the modern world, and everything in general is moving to electronics, servers and online. I felt that it would be a great idea to provide a centralized and user-friendly platform that streamlines tasks and enhances the operational efficiency on the farm. Farming in the west of Ireland needs to catch up and become more digitized, bringing modern technology solutions to this part of the country.

The Terms of reference (Scope) for my project are:

Geographic and User scope – Main target on Irish Farming operations

Technical Scope – Field Management, Weather monitoring, Farm Operations, Documentation, Technical Infrastructure (Web based app) and External API integrations.

This will involve extensive research into coding techniques to graph and display all my information for user friendliness. It will involve exploring coordinates and locations on the google Maps API and drawing polygons onto this map. It will also include extensive research into all the other Terms of reference.

5 Background and challenges faced

Digital Solutions for Modern Farming

- The agricultural sector has gone through significant changes over the last number of years. Traditional practices are quickly being augmented by digital technologies that overall improve efficiency, sustainability and profitability. Challenges that are now being faced are climate change, resource management and market volatility. This app was created to tackle these challenges with an integrated digital solution. [12]

Farm Management systems

- Keeping farm records has become harder due to the growing popularity of farming. Current systems to do this is farming mostly focus on specific farming operations, leaving gaps in other areas like keeping stock checks. FarmPulse aims to fill all these gaps.

Weather Monitoring

- Weather forecasts, especially in Ireland, can significantly impact many different areas of farming like crop planting and harvesting to livestock management. The integration of real time weather data and the OpenWeatherMap API is aiming to allow farmers to make more real time decisions based on the current forecast.

Field Mapping

- Efficient land use is crucial for maximizing productivity on the farm. Geographic Information System (GIS) technologies through the Google Maps API will enable farmers to visualise lands, set boundaries and plan activities appropriately.

6 Project Architecture

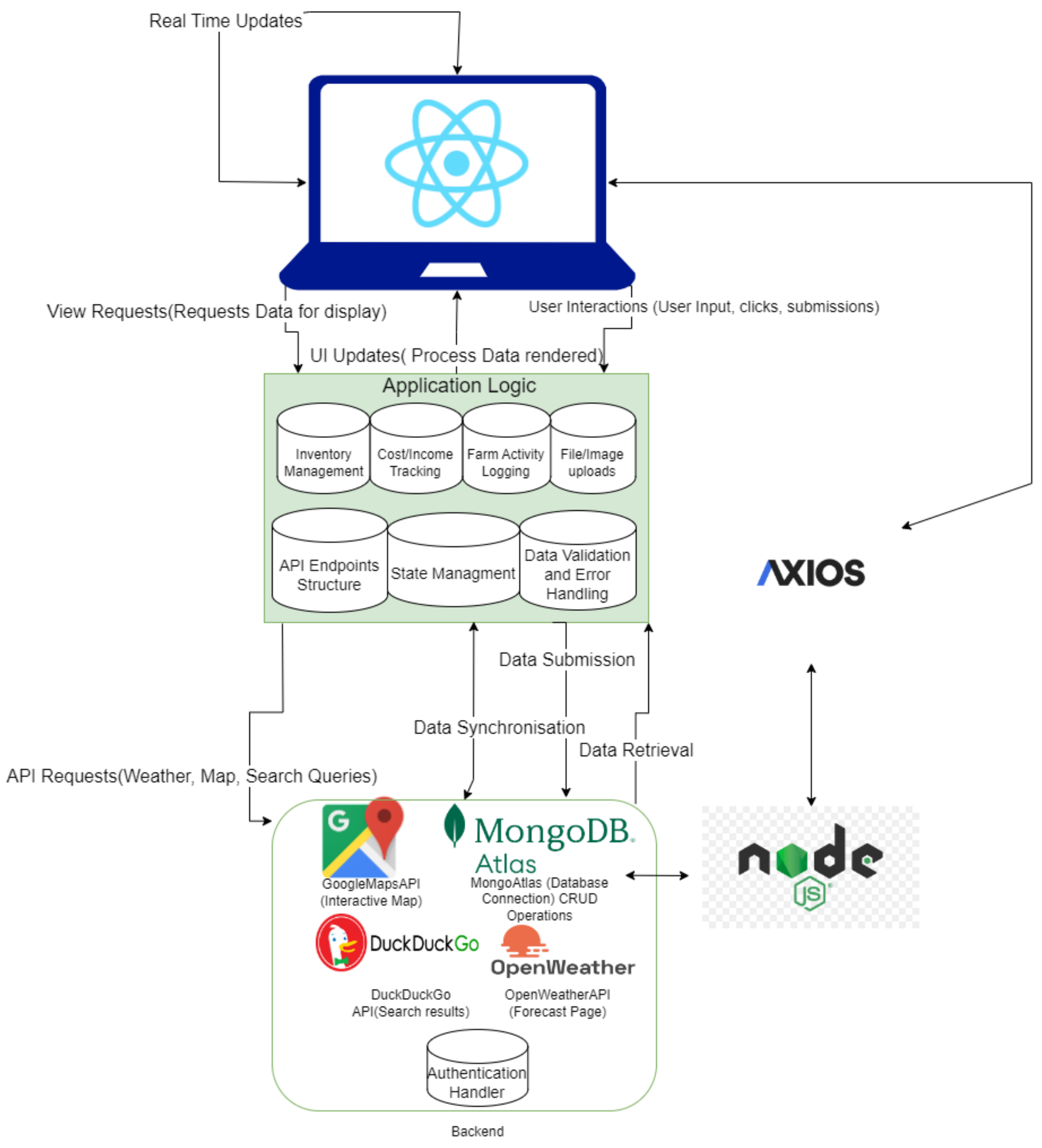
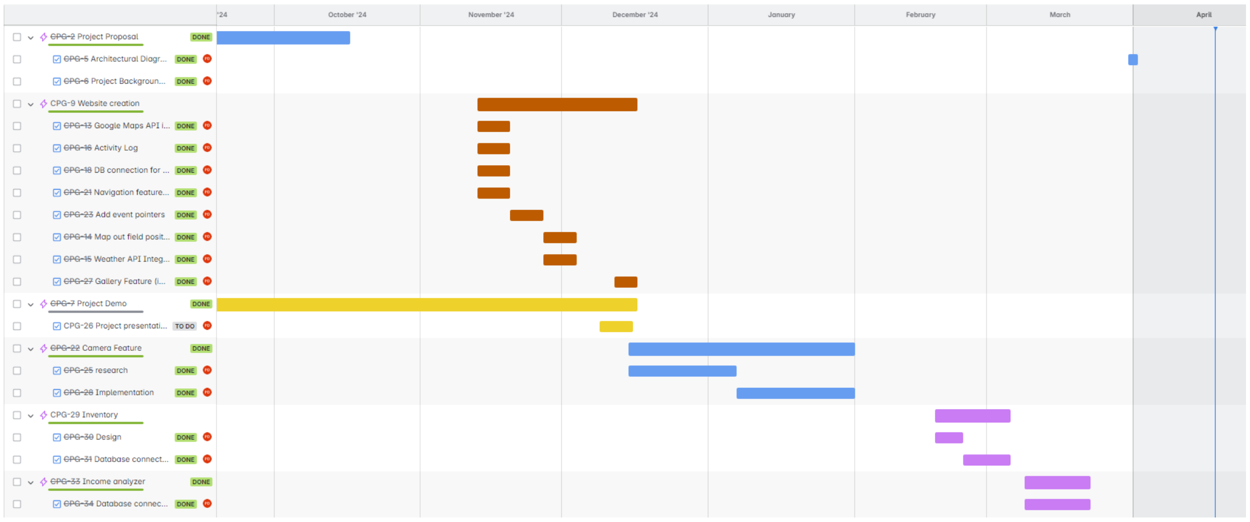

Real Time Updates
View Requests(Requests Data for display)
User Interactions (User Input, clicks, submissions)
UI Updates(Process Data rendered)
Application Logic
Inventory Management
Cost/Income Tracking
Farm Activity Logging
File/Image uploads
API Endpoints Structure
State Managment
Data Validation and Error Handling
AXIOS
Data Submission
Data Synchronisation
Data Retrieval
API Requests(Weather, Map, Search Queries)
GoogleMapsAPI (Interactive Map)
MongoDB Atlas
MongoAtlas (Database Connection) CRUD Operations
DuckDuckGo
OpenWeather
DuckDuckGo API(Search results)
OpenWeatherAPI (Forecast Page)
Authentication Handler
Backend
node JS

Figure 2: Architectural Diagram

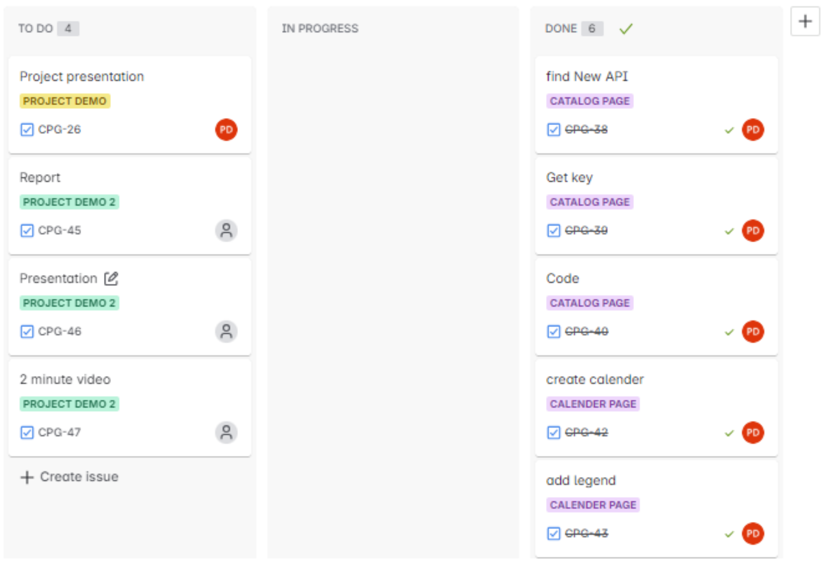
7 Project Plan

Jira was used for all the planning and execution of FarmPulse. I first started by using a beginner's guide to Jira to familiarize myself with the platform and what it does. [13] I created epic issues (large user stories) and then broke them down into smaller issues that were to be completed. I checked this project plan every 2 weeks and updated it accordingly based on the tasks I completed. I would then start a newer issue and work through that when the last one was completed after 2 weeks.



~~CPG-2~~ Project Proposal DONE
~~CPG-5~~ Architectural Diagr... DONE
~~CPG-6~~ Project Backgroun... DONE
CPG-9 Website creation
~~CPG-13~~ Google Maps API i... DONE
~~CPG-16~~ Activity Log DONE
~~CPG-18~~ DB connection for ... DONE
~~CPG-21~~ Navigation feature... DONE
~~CPG-23~~ Add event pointers DONE
~~CPG-14~~ Map out field posit... DONE
~~CPG-15~~ Weather API Integ... DONE
~~CPG-27~~ Gallery Feature (i... DONE
~~CPG-7~~ Project Demo DONE
CPG-26 Project presentati... TO DO
~~CPG-22~~ Camera Feature DONE
~~CPG-25~~ research DONE
~~CPG-28~~ Implementation DONE
CPG-29 Inventory
~~CPG-30~~ Design DONE
~~CPG-31~~ Database connect... DONE
~~CPG-33~~ Income analyzer DONE
~~CPG-34~~ Database connec... DONE

Figure 3: Jira Timeline



TO DO 4

Project presentation
PROJECT DEMO
CPG-26 PD

Report
PROJECT DEMO 2
CPG-45

Presentation
PROJECT DEMO 2
CPG-46

2 minute video
PROJECT DEMO 2
CPG-47

+ Create issue

IN PROGRESS

DONE 6 ✓

find New API
CATALOG PAGE
~~CPG-38~~ ✓ PD

Get key
CATALOG PAGE
~~CPG-39~~ ✓ PD

Code
CATALOG PAGE
~~CPG-40~~ ✓ PD

create calender
CALENDER PAGE
~~CPG-42~~ ✓ PD

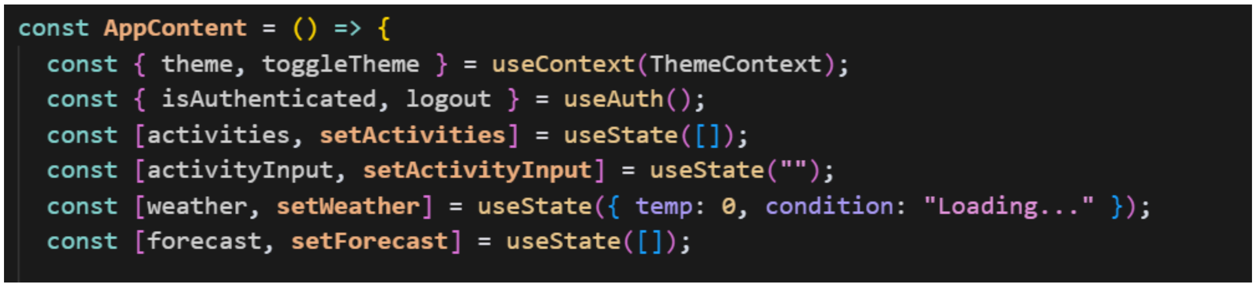
add legend
CALENDER PAGE
~~CPG-43~~ ✓ PD

Figure 4: Jira Tasks

8 Front End Architecture

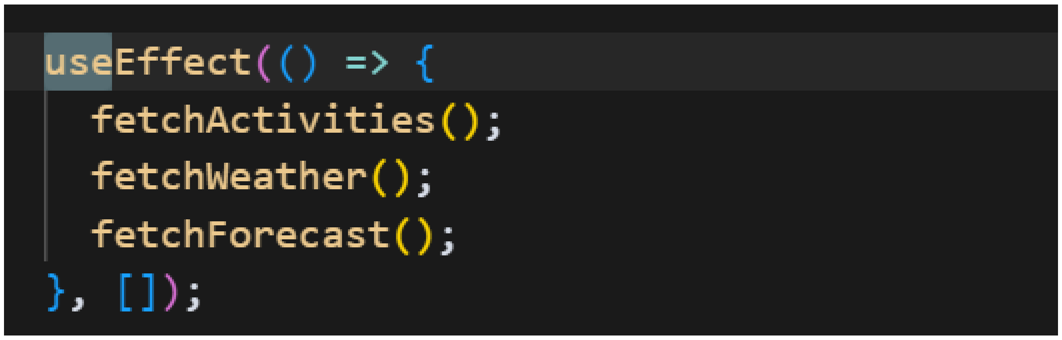
8.1 Main Application Structure (App.js)

The App.js file serves as the application's main file, handling the routing system, authentication flow, and global state management. By using React Router, the application implements client-side routing that enables seamless navigation without full page reloads and enhances the user experience.



```
const AppContent = () => {
  const { theme, toggleTheme } = useContext(ThemeContext);
  const { isAuthenticated, logout } = useAuth();
  const [activities, setActivities] = useState([]);
  const [activityInput, setActivityInput] = useState("");
  const [weather, setWeather] = useState({ temp: 0, condition: "Loading..." });
  const [forecast, setForecast] = useState([]);
```

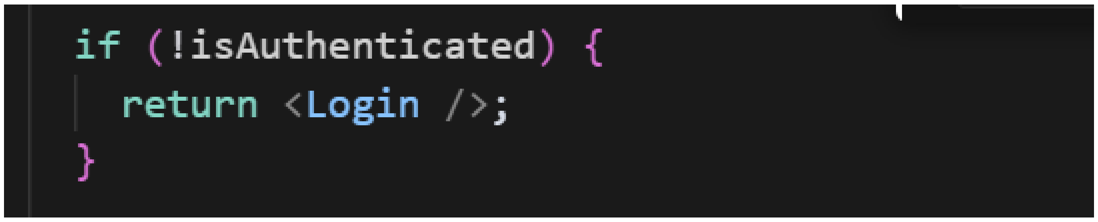
Figure 5: App content



```
useEffect(() => {
  fetchActivities();
  fetchWeather();
  fetchForecast();
}, []);
```

Figure 6: Use of react hooks

This component demonstrates the use of React Hooks (useState, useEffect, useContext) to manage state and side-effects. This use Effect is used to fetch the activities for the activity log, the weather and forecast for the front page.



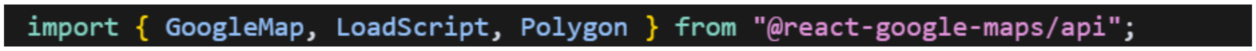
```
if (!isAuthenticated) {
  return <Login />;
}
```

Figure 7: Authentication

This authentication check provides a security layer, redirecting unauthenticated users to the login page.

8.2 Google Maps API implementation (Mapscomponent.js)

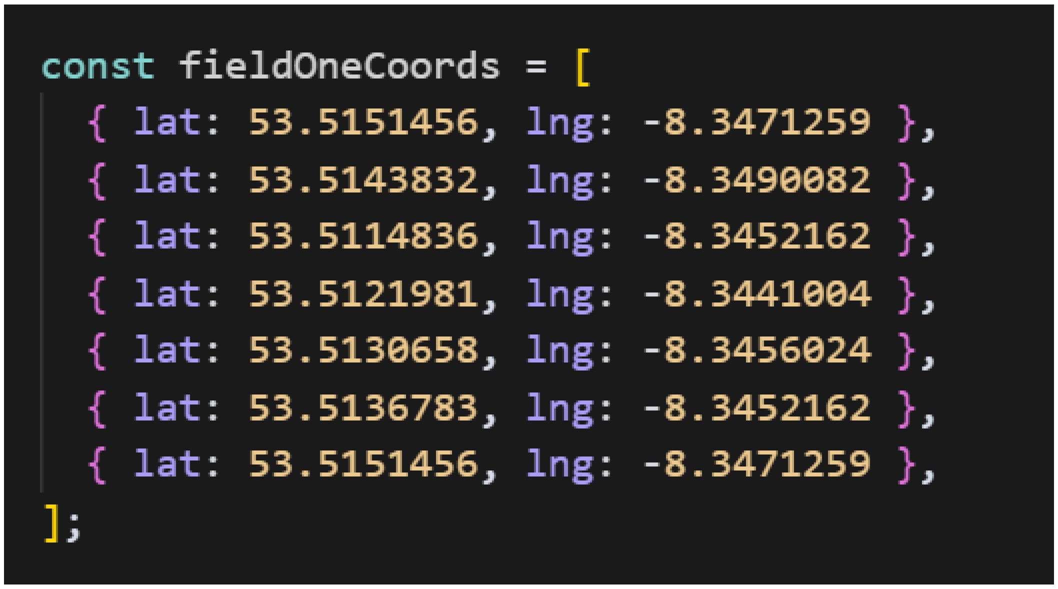
The mapcomponent.js file is the file that implements the interactive map with the GoogleMaps API. The Google Maps API documentation was a great help to implement this into my project. [1]



```
import { GoogleMap, LoadScript, Polygon } from "@react-google-maps/api";
```

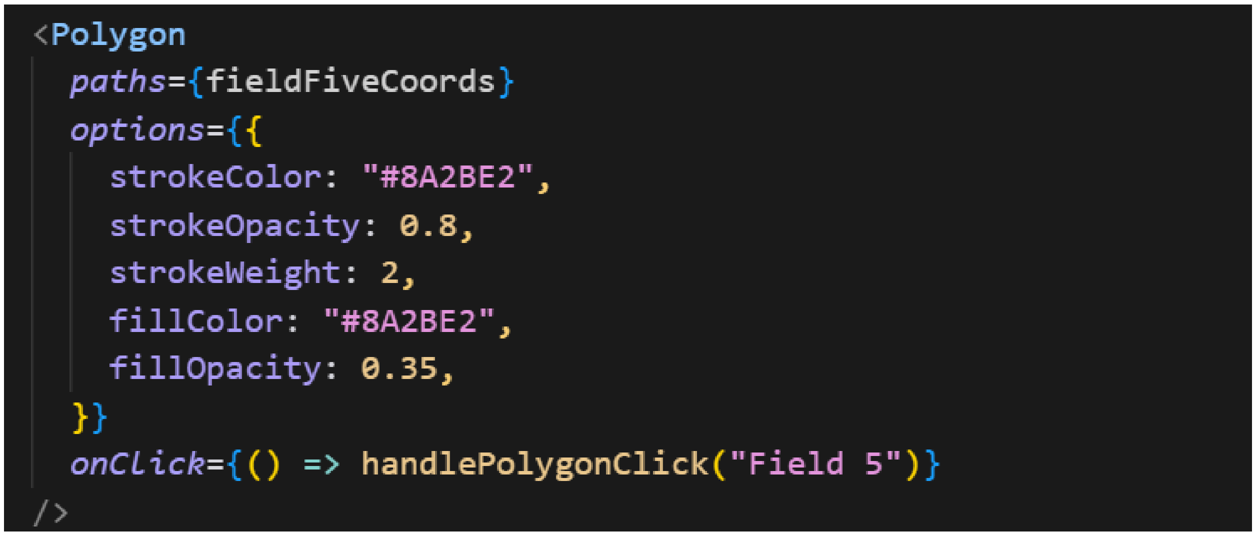
Figure 8: Google Maps API import

The GoogleMapsAPI is imported using the statement “import from @React-google-maps/api”. I imported the GoogleMap , LoadScript and Polygon components. The polygon component was an important one for this project as it allowed me to draw shapes on the map, which helped me map out the users field on the home page. The polygon implementation for field boundaries demonstrates vector-based GIS (Geographic Information System) principles, where each field is defined by precise geographic coordinates. The google maps app was a great help for me in this, as it allowed me to individually find these coordinates and draw them on the map like below. I got these coordinates from drawing them and extracting a CSV file.



```
const fieldOneCoords = [
  { lat: 53.5151456, lng: -8.3471259 },
  { lat: 53.5143832, lng: -8.3490082 },
  { lat: 53.5114836, lng: -8.3452162 },
  { lat: 53.5121981, lng: -8.3441004 },
  { lat: 53.5130658, lng: -8.3456024 },
  { lat: 53.5136783, lng: -8.3452162 },
  { lat: 53.5151456, lng: -8.3471259 },
];
```

Figure 9: Field coordinates



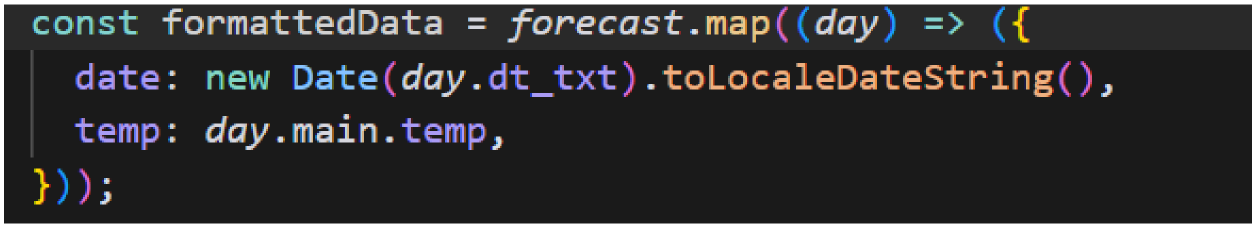
```
<Polygon
  paths={fieldFiveCoords}
  options={{
    strokeColor: "#8A2BE2",
    strokeOpacity: 0.8,
    strokeWeight: 2,
    fillColor: "#8A2BE2",
    fillOpacity: 0.35,
  }}
  onClick={() => handlePolygonClick("Field 5")}
/>
```

Figure 10: Polygon feature

Each polygon had their own paths, which was used to set things like colour, weight and fill colour. These polygons could also be clicked on by a user, which would send them to a separate page with a picture of their fields.

8.3 Weather Integration (ForecastPage.js)

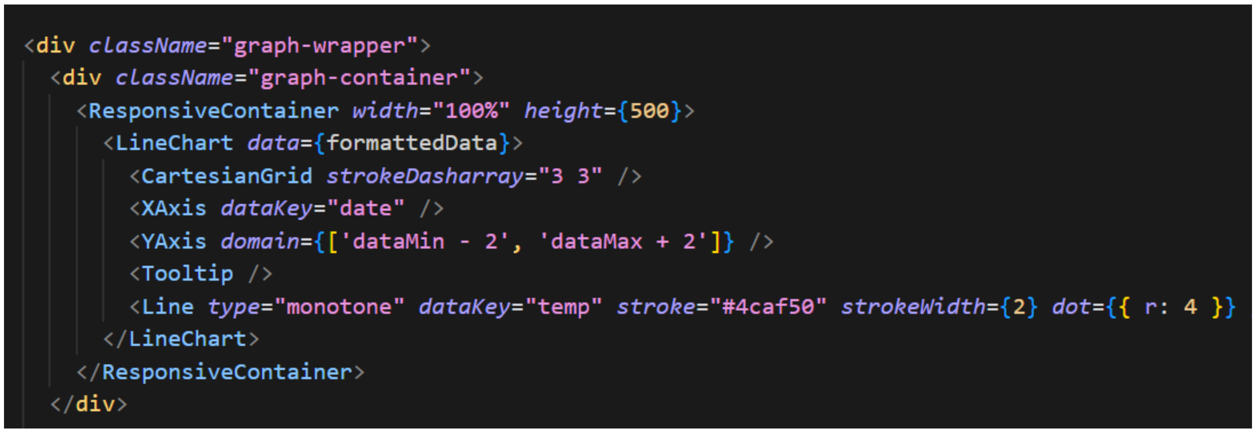
Weather information for farming is probably one of the most important things, as it allows farmers to plan and execute different tasks when the weather is suited to do so. This page allows a good visualization of metrological data , presented through text and graphical features.



```
const formattedData = forecast.map((day) => ({
  date: new Date(day.dt_txt).toLocaleDateString(),
  temp: day.main.temp,
}));
```

Figure 11: Formatted Data

This creates a new array called formattedData which transforms each day object from the forecast array, It extracts and formats the data string into a human readable date. It also gets the temperature from day.main.temp.



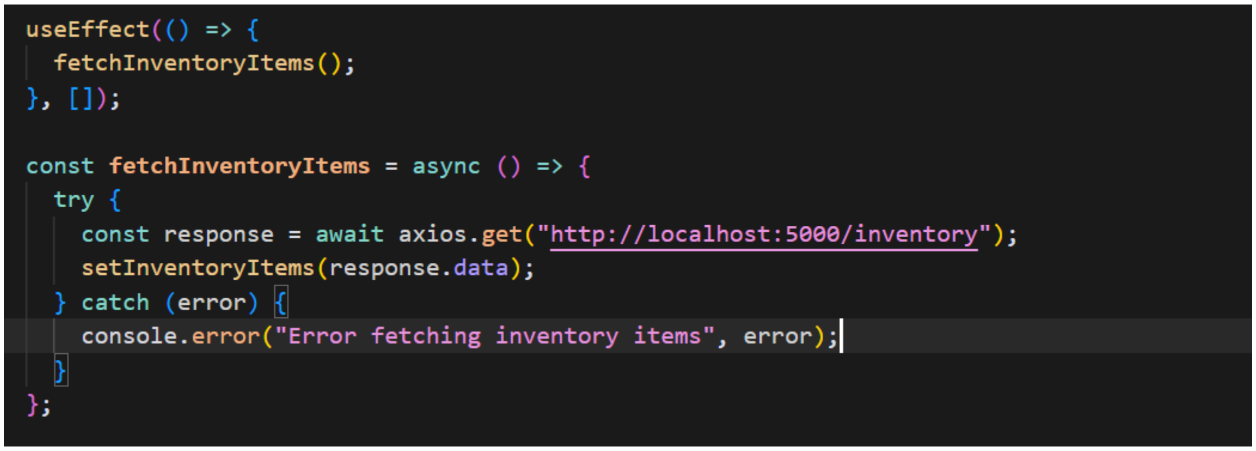
```
<div className="graph-wrapper">
  <div className="graph-container">
    <ResponsiveContainer width="100%" height={500}>
      <LineChart data={formattedData}>
        <CartesianGrid strokeDasharray="3 3" />
        <XAxis dataKey="date" />
        <YAxis domain={['dataMin - 2', 'dataMax + 2']} />
        <Tooltip />
        <Line type="monotone" dataKey="temp" stroke="#4caf50" strokeWidth={2} dot={{ r: 4 }}
      </LineChart>
    </ResponsiveContainer>
  </div>
```

Figure 12: Line Chart with recharts

A line chart is then rendered using Recharts, which is a react library. [10] Firstly, the chart is made responsive to the screen size. The formatted data is the source of data to be on this line chart. A gridded background is added for better readability and enhanced user friendliness. Tooltip is enabled to show details when a user hovers over a point on the graph. <Line> then plots the actual temperature line, and it is smoothened using Monotone. The forecast is fetched through the OpenWeatherMap API. [2]

8.4 Inventory Management (InventoryPage.js)

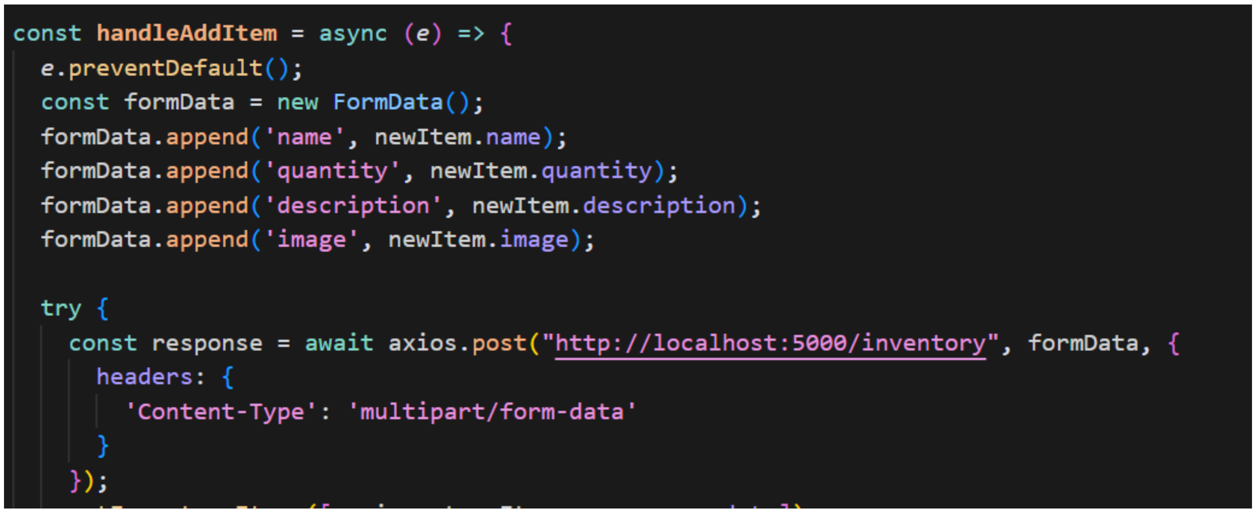
The inventory was stored using MongoDB Mongo Atlas. [3] This inventory page shows CRUD operations using react and Axios [6] for API communication. It can handle complex data, including media-uploads thanks to Multer.



```javascript
useEffect(() => {
  fetchInventoryItems();
}, []);

const fetchInventoryItems = async () => {
  try {
    const response = await axios.get("http://localhost:5000/inventory");
    setInventoryItems(response.data);
  } catch (error) {
    console.error("Error fetching inventory items", error);
  }
};
```

Figure 13: GET Request Inventory

UseEffect is used to make a GET request which retrieves the data Inventory when the component mounts. Axios is used for the GET request.



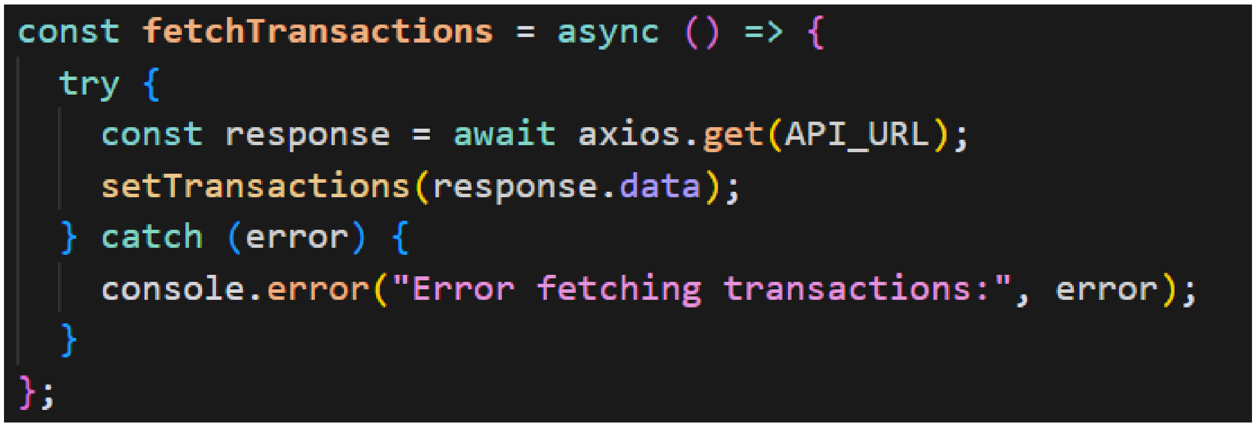
```javascript
const handleAddItem = async (e) => {
  e.preventDefault();
  const formData = new FormData();
  formData.append('name', newItem.name);
  formData.append('quantity', newItem.quantity);
  formData.append('description', newItem.description);
  formData.append('image', newItem.image);

  try {
    const response = await axios.post("http://localhost:5000/inventory", formData, {
      headers: {
        'Content-Type': 'multipart/form-data'
      }
    });
```

Figure 14: POST Request

A form collects item data, like name, quantity, description and image. This is sent as FormData using a post request and is saved into the Atlas Database. The same concept is used for editing existing items; however this uses a PUT request. Images are rendered by using the imageURL provided by the backend.

8.5 Financial Tracking (CostIncomePage.js)

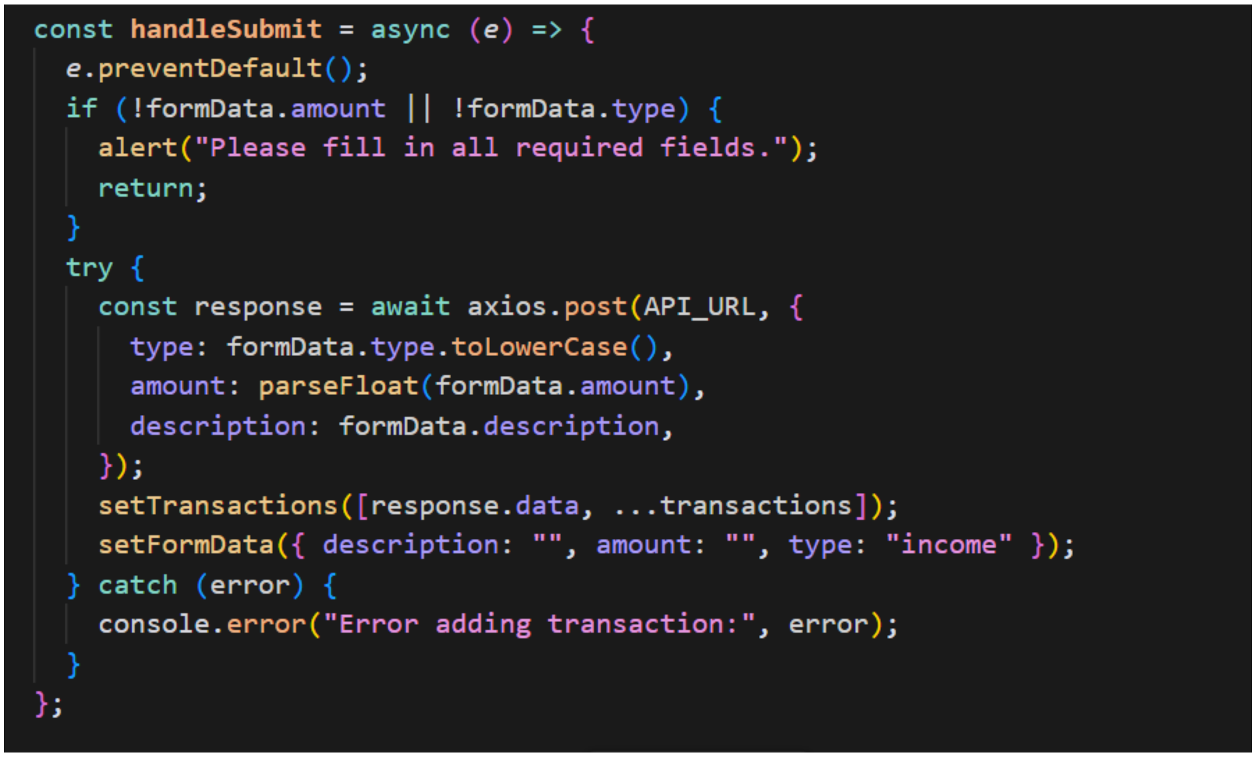
This page is important to farmers, as the cost of stock, animals and machinery has increased. This provides farmers with important financial information, specifically their cost and income, with a visual element for quick and easy clarification of costs.



```
const fetchTransactions = async () => {
  try {
    const response = await axios.get(API_URL);
    setTransactions(response.data);
  } catch (error) {
    console.error("Error fetching transactions:", error);
  }
};
```

Figure 15: Fetch Transactions

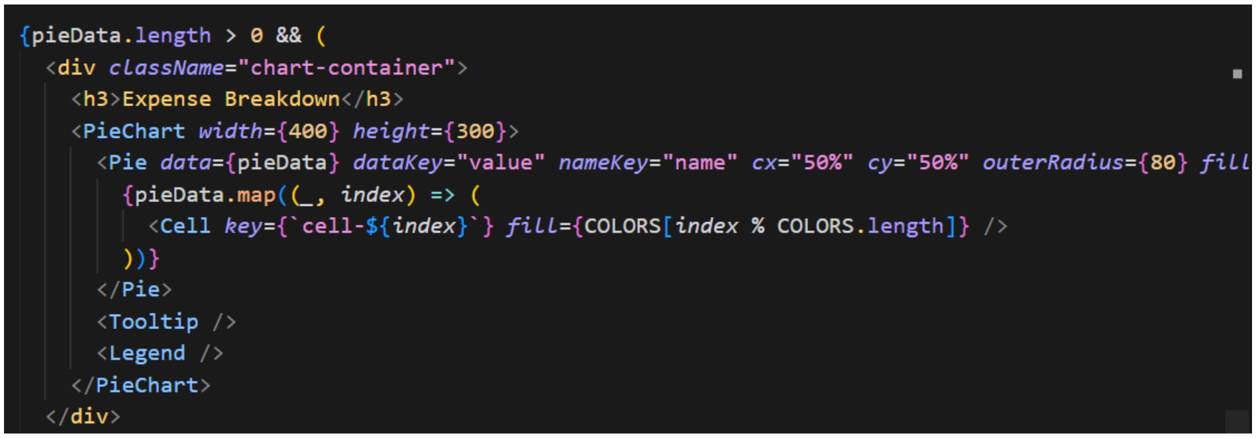
The useEffect hook ensures that the transaction data is automatically loaded when the page is opened. The axios.get(API_URL) ensures uses a GET request to retrieve all the existing transactions. The result is stored in the transactions state, allowing it to be rendered and processed within the component. This ensures the page is always displaying the most recent and current transactions from the backend.



```
const handleSubmit = async (e) => {
  e.preventDefault();
  if (!formData.amount || !formData.type) {
    alert("Please fill in all required fields.");
    return;
  }
  try {
    const response = await axios.post(API_URL, {
      type: formData.type.toLowerCase(),
      amount: parseFloat(formData.amount),
      description: formData.description,
    });
    setTransactions([response.data, ...transactions]);
    setFormData({ description: "", amount: "", type: "income" });
  } catch (error) {
    console.error("Error adding transaction:", error);
  }
};
```

Figure 16: Submit Income/Expenditure

This function handles the form submission whenever a user adds a new income or expenditure entry. It validates all the inputs, coverts the amount to a number and then sends the transactions to the backend with a POST request. After a successful response is triggered, the new entry is added to the top of the transaction list. This provides real time feedback to the user a keeps the list updated without having to refresh the page.



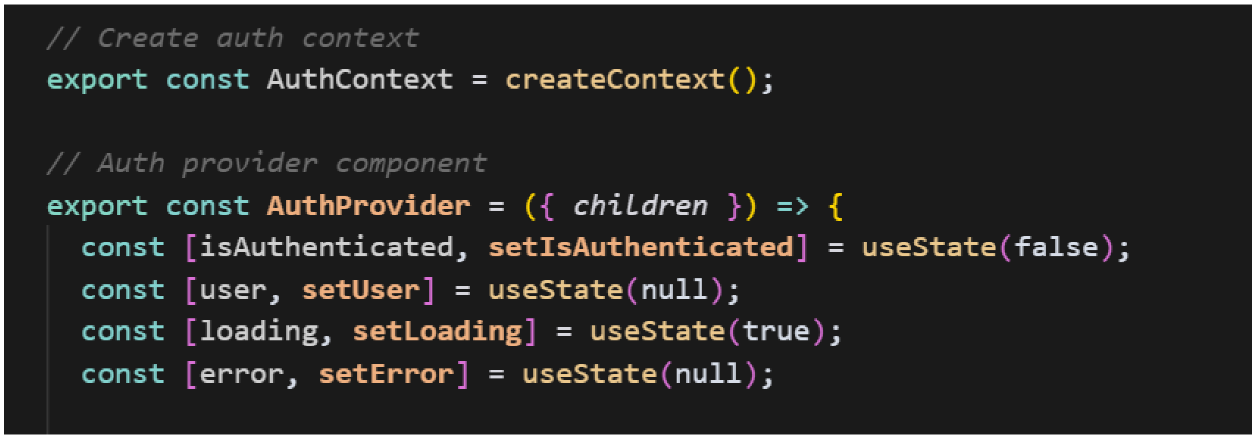
```
{pieData.length > 0 && (
  <div className="chart-container">
    <h3>Expense Breakdown</h3>
    <PieChart width={400} height={300}>
      <Pie data={pieData} dataKey="value" nameKey="name" cx="50%" cy="50%" outerRadius={80} fill
        {pieData.map((_, index) => (
          <Cell key={`cell-${index}`} fill={COLORS[index % COLORS.length]} />
        ))}
      </Pie>
      <Tooltip />
      <Legend />
    </PieChart>
  </div>
```

Figure 17: Pie chart using recharts

Again, recharts is used, this time for a piechart. The transactions are grouped and the total cost is calculated. The piechart visually displays how the expenses are distributed across different categories, using different colours for each section. This provides a very clear and insightful visual summary of farm expenditure patterns for the user, helping identify and manage costs effectively.

8.6 Authentication System (AuthContext.js and Login.js)

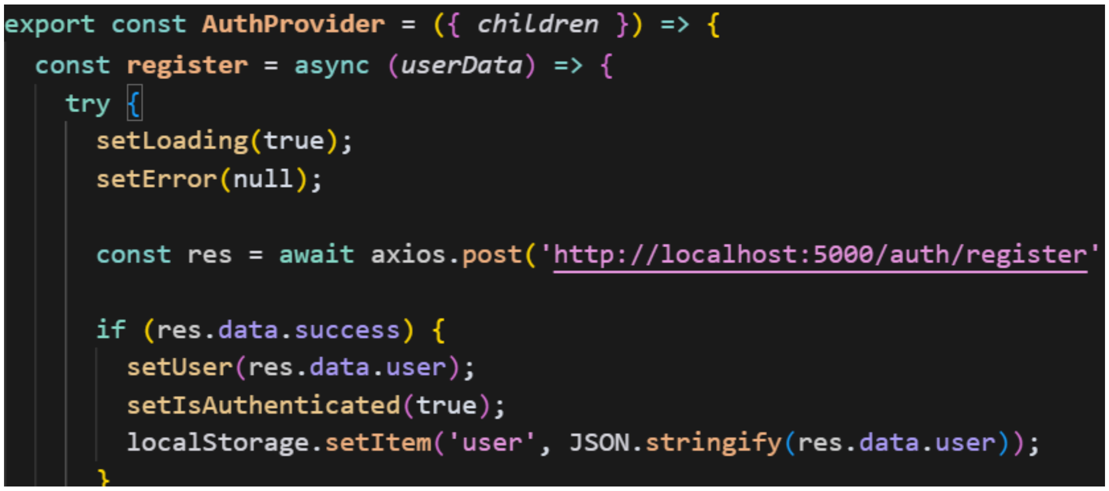
The user authentication system in this application is built using React Context for centralized state management. An AuthContext is created and provided through the AuthProvider component, which wraps around the app to supply global access to authentication-related data such as the user object, login status, and error/loading states.



```
// Create auth context
export const AuthContext = createContext();

// Auth provider component
export const AuthProvider = ({ children }) => {
  const [isAuthenticated, setIsAuthenticated] = useState(false);
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);
```

Figure 18: Auth provider

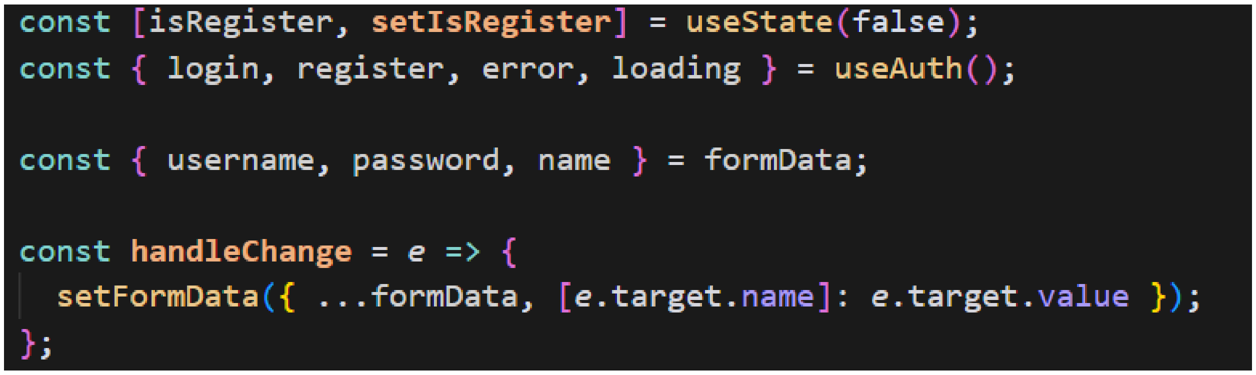
When a user logs in or registers, a POST request is sent to the appropriate backend route (/auth/login or /auth/register). On successful authentication, the user data is saved in both the local state and localStorage to maintain session persistence even after page refreshes.



```
export const AuthProvider = ({ children }) => {
  const register = async (userData) => {
    try {
      setLoading(true);
      setError(null);

      const res = await axios.post('http://localhost:5000/auth/register'

      if (res.data.success) {
        setUser(res.data.user);
        setIsAuthenticated(true);
        localStorage.setItem('user', JSON.stringify(res.data.user));
      }
```

Figure 19: Register

The login interface is dynamically designed to handle both login and registration using a toggle (isRegister) which adjusts the form fields and button label accordingly.



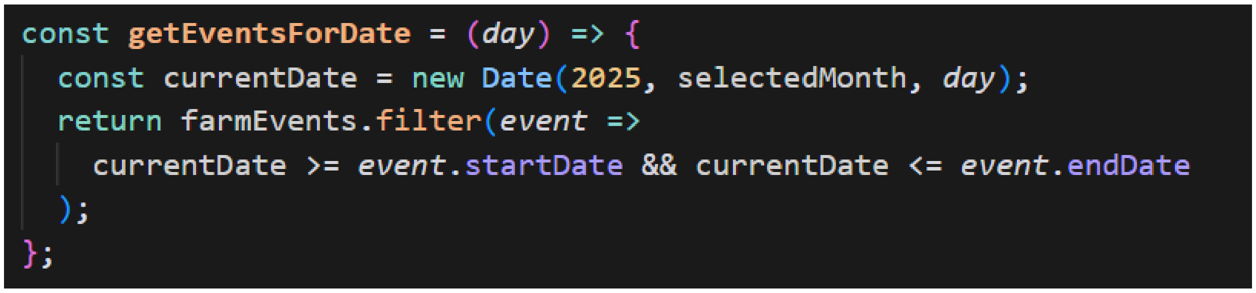
```
const [isRegister, setIsRegister] = useState(false);
const { login, register, error, loading } = useAuth();

const { username, password, name } = formData;

const handleChange = e => {
  setFormData({ ...formData, [e.target.name]: e.target.value });
};
```

Figure 20: Login Interface

This allows for a user-friendly, single-page entry point into the app. Error messages from failed attempts are displayed inline, and a loading state is shown while the request is processed, ensuring responsive feedback. Overall, this system provides a clean and efficient way for users to authenticate, register, and remain signed in across sessions.

8.7 Job Planning (Calendarpage.js)

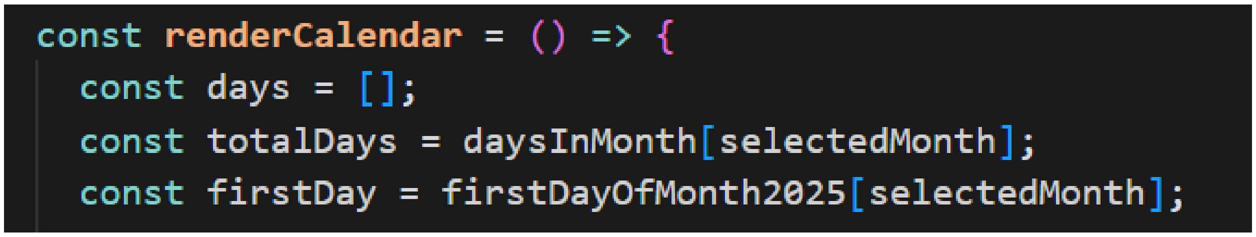
Job planning is one of the most important features in farming. Certain days and seasons calls for certain jobs to be done, like slurry spreading, lambing season and calving season. The calendar page provides a fully interactive farm calendar that allows users to view, navigate by month and add new events tied to specific dates.



```
const getEventsForDate = (day) => {
  const currentDate = new Date(2025, selectedMonth, day);
  return farmEvents.filter(event =>
    currentDate >= event.startDate && currentDate <= event.endDate
  );
};
```

Figure 21: getEventsforDate

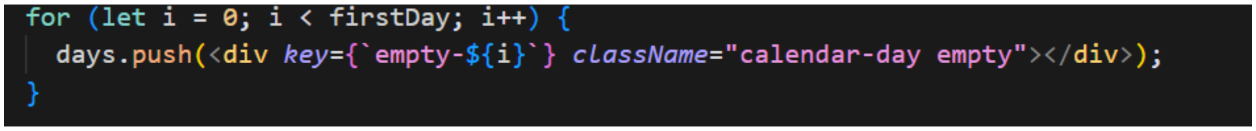
The get events for Date function is the heart of the event logic in the calender component. For each day in the month, it checks whether the date falls in the range of any event (eg – Lambing season in March – April) and displays them. This makes the calendar dynamic and interactive instead of being a static grid.



```
const renderCalendar = () => {
  const days = [];
  const totalDays = daysInMonth[selectedMonth];
  const firstDay = firstDayOfMonth2025[selectedMonth];
```

Figure 22: Render Calendar

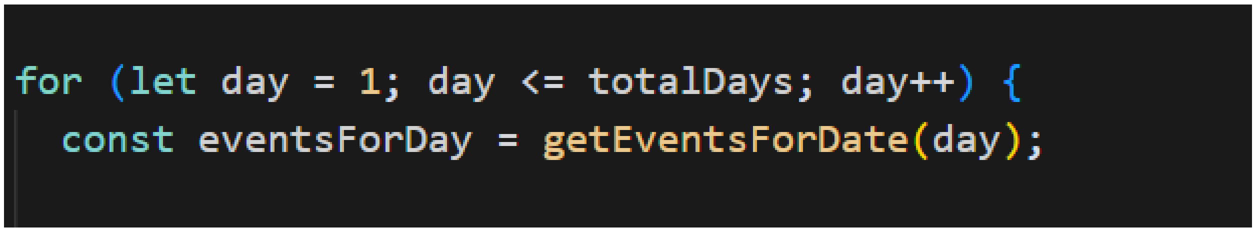
The render calendar function generates the calendar. Days is an empty array cell that will hold all day cells for the calendars grid. const totalDays = daysInMonth[selectedMonth] line gets the actual number of days in the selected month and used to loop through the actual days. const firstDay = firstDayOfMonth[selectedMonth] looks up what day the first day of the month is and offsets the calender so it all aligns up correctly.



```
for (let i = 0; i < firstDay; i++) {
  days.push(<div key={`empty-${i}`} className="calendar-day empty"></div>);
}
```

Figure 23: Empty filler cells

The loop above adds in empty cells at the start of the month that doesn't have a date. Eg – If the 1st falls on a Wednesday, the Sunday, Monday and Tuesday will be empty. It adds empty div elements to do this.



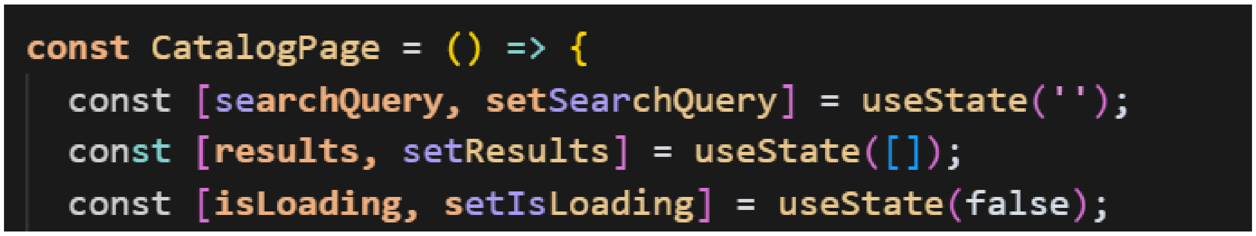
```
for (let day = 1; day <= totalDays; day++) {
  const eventsForDay = getEventsForDate(day);
```

Figure 24: Get events on a certain day

This loop runs once for each day of the month. getEventsForDate(day) is called to check if any farm events happen on that day. It then returns an array of events that fall on the specific date.

8.8 Product Searching (CatalogPage.js)

This component allows users to search for farm products using Google Results with SerpAPI, a comprehensive solution for seamless search engine data collection, handling proxy management, unblocking, and parsing with ease. [5]

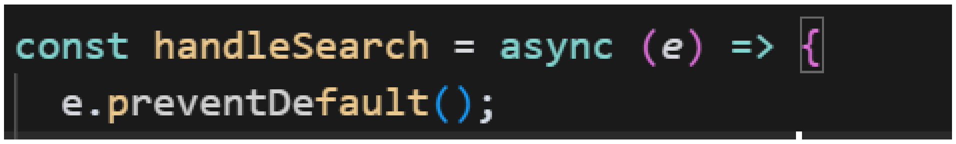


```
const CatalogPage = () => {
  const [searchQuery, setSearchQuery] = useState('');
  const [results, setResults] = useState([]);
  const [isLoading, setIsLoading] = useState(false);
```

Figure 25: Search logic

This code manages the search input, the results that are shown and the loading state. Chat GPT [4] helped me implement this feature by suggesting SerpAPI, as DuckDuckGo API I first wanted to use required a paid subscription. It also gave me the serpAPI docs webpage link, which was very helpful. [5]

An API key was also obtained from the Serp API website and is gotten from my .env file that contains my other API keys.



```
const handleSearch = async (e) => {
  e.preventDefault();
```

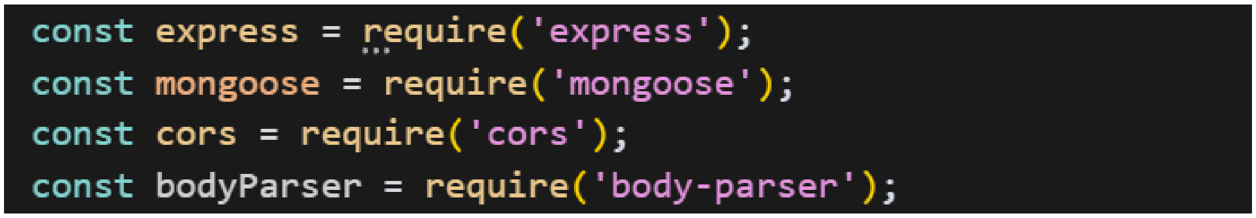
Figure 26: Error handling

When the user searches, a request is built to google through Serp API. Various errors are also handled here, including 401 (bad API key) and 429 (too many requests) errors. The results are then saved and displayed. If a search fails or nothing shows up, there is fallback results also rendered.

9 Back End Architecture

9.1 Back End server implementation (server.js)

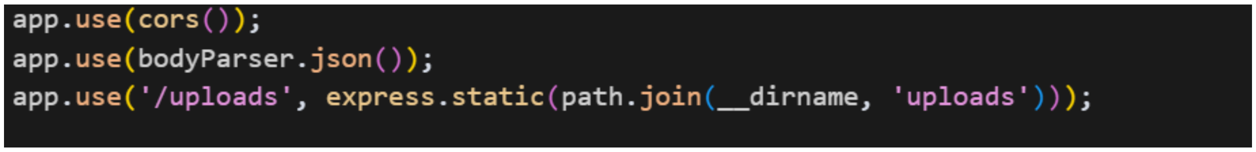
Server.js is used to setup the express app and import all the tools needed. It also contains the user routes and the port that the server starts running on. Node.js is used. [11]



```
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
const bodyParser = require('body-parser');
```

Figure 27: Backend Technologies

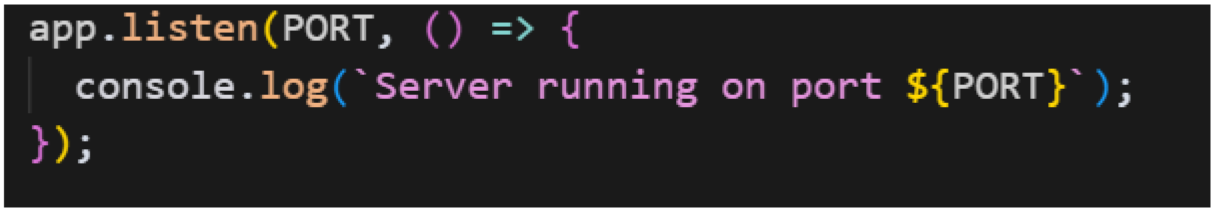
Express builds my API. Mongoose connects to my mongo database. Cors allows the frontend application talk to the backend. Body-parser reads JSON format that is sent from the frontend.



```
app.use(cors());
app.use(bodyParser.json());
app.use('/uploads', express.static(path.join(__dirname, 'uploads')));
```

Figure 28: Serve uploaded images

This code above allows the server to serve the uploaded images that a user has uploaded and displays them on the webpage.



```
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Figure 29: Server Port

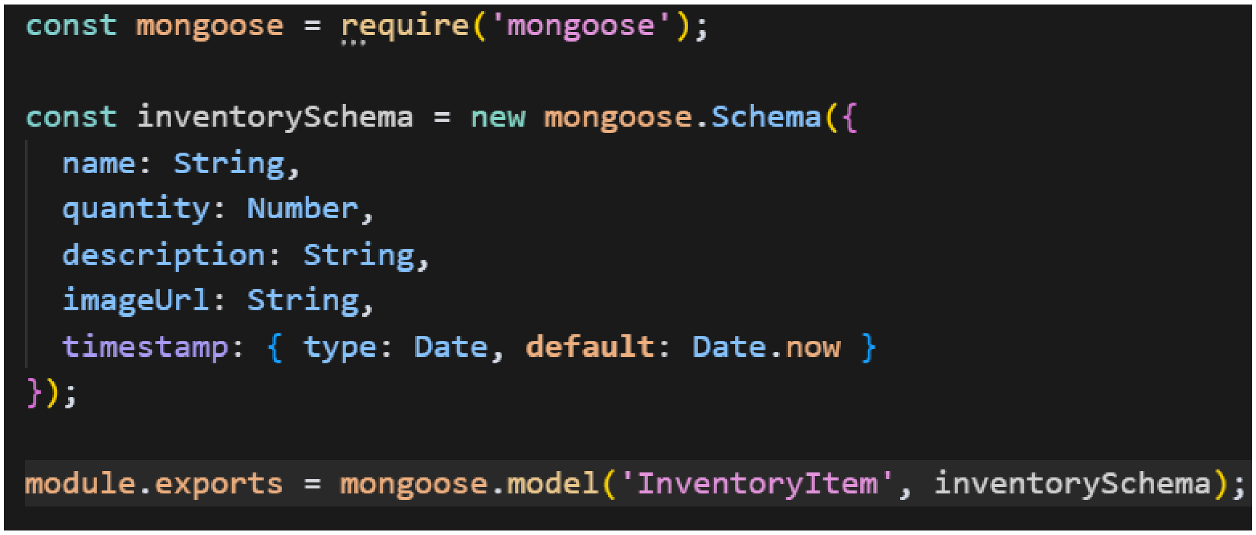
The backend is then started on whatever port I have defined.

9.2 Models

The model's folder contains all the schemas that is made for the backend. There is a schema for:

1. Activity (Activity.js)
2. Cost Income (CostIncome.js)
3. Inventory (Inventory.js)
4. User (User.js)

The MongoDB schemas define the fields that will be required to be populated in my MongoDB.



```
const mongoose = require('mongoose');

const inventorySchema = new mongoose.Schema({
  name: String,
  quantity: Number,
  description: String,
  imageUrl: String,
  timestamp: { type: Date, default: Date.now }
});

module.exports = mongoose.model('InventoryItem', inventorySchema);
```

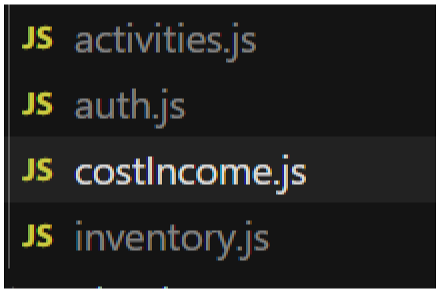
Figure 30: A Mongo Schema (Inventory)

For example, my inventory schema above contains the name of the item, which is a string, the quantity the farmer has which is a number, the description as a string and the imageUrl as a string thanks to multer. Multer is a node.js middleware for handling multipart/form-data, which is primarily used for uploading files. [8] There is also a timestamp to let the farmer know what time and date the item was added. The model is then registered with mongoose.

All the backend Models files follow the exact same flow to create the models for mongo.

9.3 Routes

The routes folder in my backend contains all the core API endpoints that define how your application handles various requests.



```
activities.js
auth.js
costIncome.js
inventory.js
```

Figure 31: Routes Architecture

Activities.js manages routes related to logging and retrieving farm activities. costIncome.js manages the routes that deal with tracking the expenses and income. Inventory.js deals with the inventory related routes, such as adding, updating and fetching inventory items. This structure keeps my routes organized which is easier for testing.

Auth.js handles authentication routes such as the user registration and login. It also uses bcrypt to hash passwords for security purposes. The code below was gathered from bcrypt.js npm webpage. [9]

```
const bcrypt = require('bcrypt');
```

```
const salt = await bcrypt.genSalt(10);
const hashedPassword = await bcrypt.hash(password, salt);
```

Figure 32: Password Hashing

The code above takes the plain password and the generated salt. It then returns a hashed version of the password that is safe to store in the MongoDB. This means the original password can’t be obtained from the hashed one in the database.

```
const isMatch = await bcrypt.compare(password, user.password);
```

Figure 33: Hashed Password comparison

During login, the hashed password is compared with the stored hash. If it is a match, the user is loggedIn. If it doesn't, then the user will be prompted with an "invalid credentials" popup.

9.4 Utilities

The utilities folder contains the helper modules that were used for testing and deployment throughout my build process.

It contains a seedUser.js file for testing in the start of my development for the Login function. This was used with bcrypt for testing hashing user passwords also, which is used in my auth.js file.

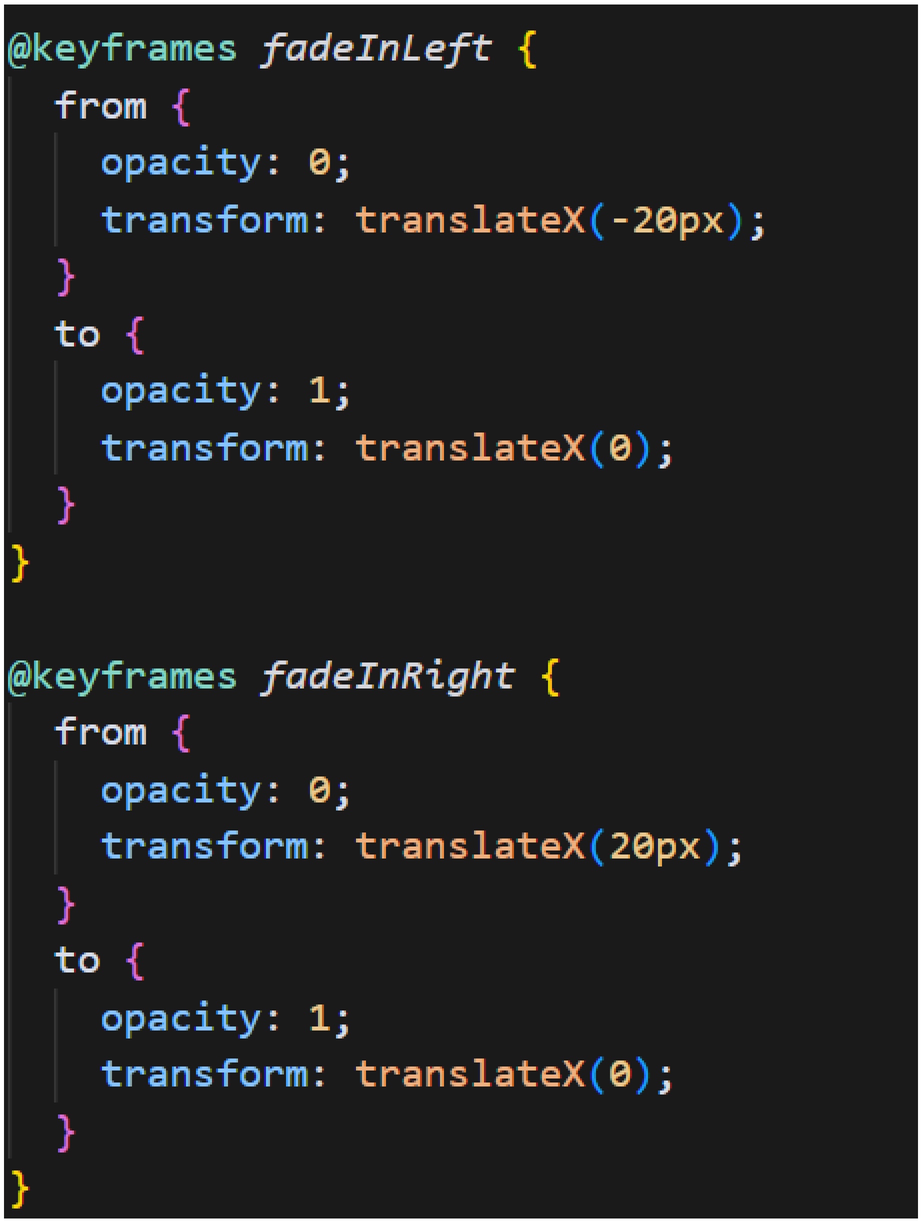
10 Website Design

All the pages were designed using CSS code. I used a combination of flexbox layouts with a consistent colour theme and smooth animations.

The green coloured palette (#4caf50) was used to establish a natural tone, while also keeping it relevant to agriculture and farming in whole. Flexboxes are used for ensuring content is always well aligned and responsive through various screen sizes.

Visual elements like buttons and containers were styled with shadows, transitions and some hover effects to make interactions feel more intuitive for the user.

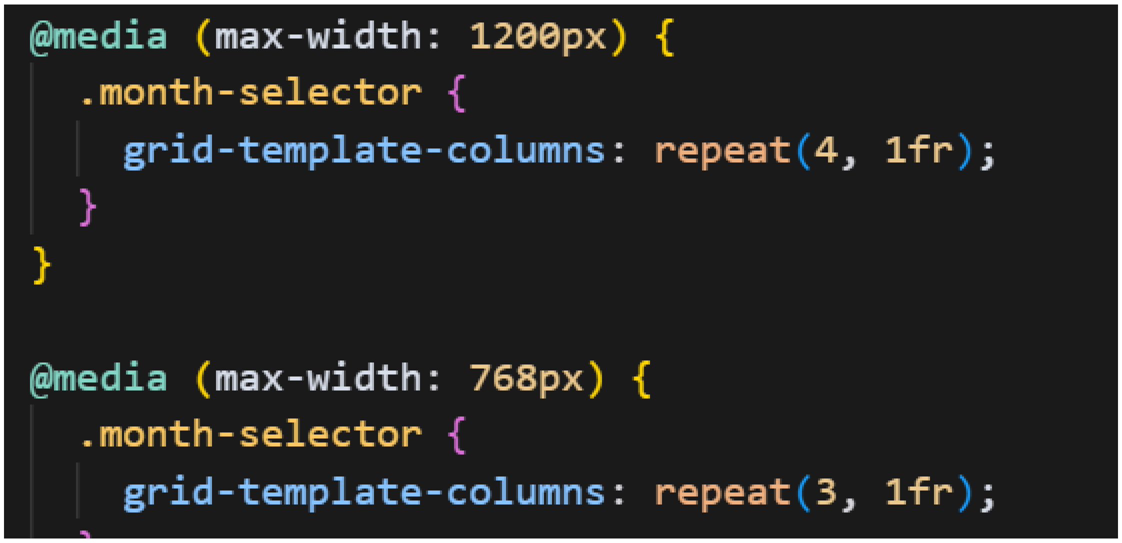
Animations like fadeIn, fadeInLeft and fadeInRight helped the website feel a lot more dynamic and polished when the components had loaded.



```css
@keyframes fadeInLeft {
  from {
    opacity: 0;
    transform: translateX(-20px);
  }
  to {
    opacity: 1;
    transform: translateX(0);
  }
}

@keyframes fadeInRight {
  from {
    opacity: 0;
    transform: translateX(20px);
  }
  to {
    opacity: 1;
    transform: translateX(0);
  }
}
```

Figure 34: CSS Animations

Media queries were also used, for example in the calendar page below, as it allowed the calendar to adapt to the different screen sizes. The number of columns in each month and the size of the calendar's days adjusted depending on screen width.



```css
@media (max-width: 1200px) {
  .month-selector {
    grid-template-columns: repeat(4, 1fr);
  }
}

@media (max-width: 768px) {
  .month-selector {
    grid-template-columns: repeat(3, 1fr);
```

Figure 35: Media Queries

11 Choice of Tools and Frameworks

For FarmPulse, react [7] alongside React router, MongoDB, and Axios was used instead of next.js. The decision was based on multiple reasons.

When I first started the project, I had more experience using these technologies ahead of next.js. This allowed me to do a significant amount at the start of the project with these technologies, and they were perfect for the application I wanted to make.

React Router allowed me to define and manage my routes completely on the client side which gave me full flexibility over all the navigation logic in the project, without having to rely on the server-rendered pages.

Axios was very sufficient for making promise-based HTTP client requests from my react frontend to my express backend, and this allowed for separation between the two but still being able to easily work together. Ultimately, I wanted my frontend and backend completely separate as it was easier for testing and future developments if needed.

MongoDB integrates nearly effortlessly with express and supports flexible schemas making it well suited for storing data that is taken from FarmPulse.

Overall, these tools made the application modular, easy to test and scalable for future developments.

However, one downside to this that I do acknowledge is no server-side rendering or automatic routing is present like it is in next.js, and this could of sped up the process slightly in building the application if I wanted the frontend and backend together in one framework.

12 Ethics

There are multiple Ethical Considerations that would need to be taken within a project like FarmPulse.

1. Data Privacy – Farmers data should be used and stored securely while also allowing them transparency on where it is going. Farm Pulse ensures the users passwords are always secure by hashing them using bcrypt.
2. Inclusive Access – The platform should be designed to be accessible to all smaller farmers to bigger and more industrialised farmers. It should provide a digital solution to all. FarmPulse ensures this web app can be accessed on a range of devices, like on a phone's browser and their laptop on many different web engines.
3. Environmental Sustainability – How farmers crops are stored, kept track of and used is something that FarmPulse had to combat. Our Inventory and Activity log feature ensures that farmers always have a way to check, add and delete stock to ensure all of it is being used, without buying extra unnecessary products or spreading extra manure that may harm the environment even more.

13 Conclusion

FarmPulse is a comprehensive application for farmers across the island of Ireland, allowing farmers to manage all their farm operations on one simple to use web application.

The backend of FarmPulse is a secure and well-organized server-side system built using Express.js and MongoDB. It includes user authentication with password hashing using bcrypt, ensuring data and passwords are securely handled. The system also features the ability to seed a default user, which speeds up the startup process for new farmers.

The frontend of FarmPulse contains the home page, with an interactive farm map giving farmers a clean and user-friendly view of their land. It also allows them to log activities and check the weather at the current moment of viewing.

The forecast page presents a clean and well used graph, which shows weather trends over the week, with symbols also for easier viewing.

The inventory page allows farmers to keep track of their stock and upload images to these items.

The camera page allows the farmer to connect their camera up, which can be used for simply taking photos or for CCTV purposes.

The cost and income page allows the farmer to manage income and expenditure easily, with a pie chart providing an intuitive interface to keep track of these costs.

The catalog page allows the farmer to search the wider web for products they may be short of – eg Meal or maybe a new Massey Ferguson.

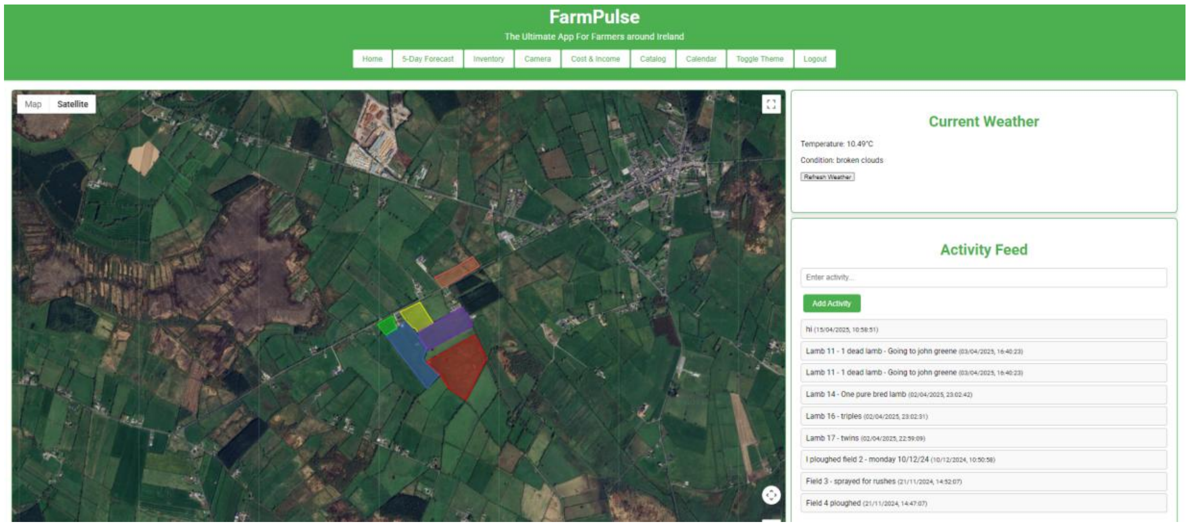
The Calendar page gives a full 2025 calendar which the farmer can use to plan activites around certain times of the year.

Toggle theme allows the farmer to use a dark theme if it is preferred.

The user also has to login before they can access these features with the Login Page, and the logout button allows the user to logout of their account.

Overall, I feel that FarmPulse was a success in bringing an easy to use one stop application for farmers.

Below are some examples of some of the pages on FarmPulse.



Current Weather

Temperature: 10.49°C

Condition: broken clouds

Activity Feed

hi (15/04/2025, 10:58:51)

Lamb 11 - 1 dead lamb - Going to john greene (03/04/2025, 16:40:23)

Lamb 11 - 1 dead lamb - Going to john greene (03/04/2025, 16:40:23)

Lamb 14 - One pure bred lamb (02/04/2025, 23:02:42)

Lamb 16 - triples (02/04/2025, 23:02:31)

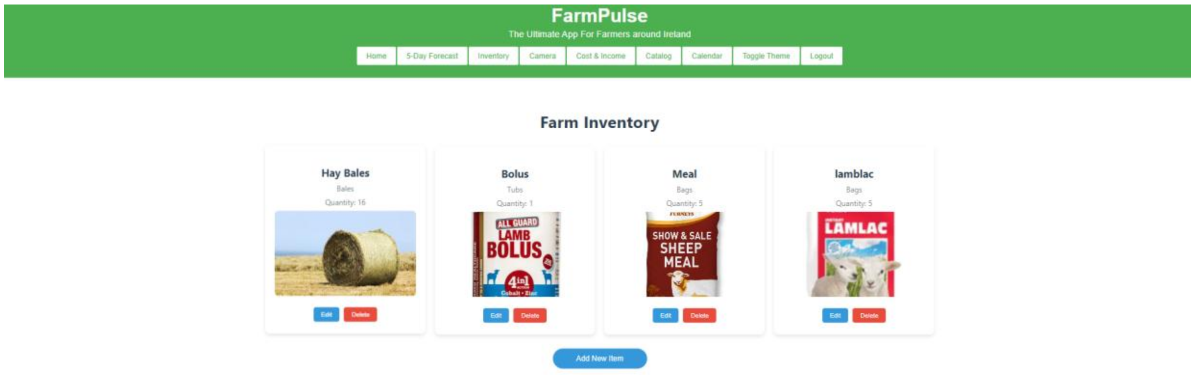
Lamb 17 - twins (02/04/2025, 22:59:09)

I ploughed field 2 - monday 10/12/24 (10/12/2024, 10:50:58)

Field 3 - sprayed for rushes (21/11/2024, 14:52:07)

Field 4 ploughed (21/11/2024, 14:47:07)

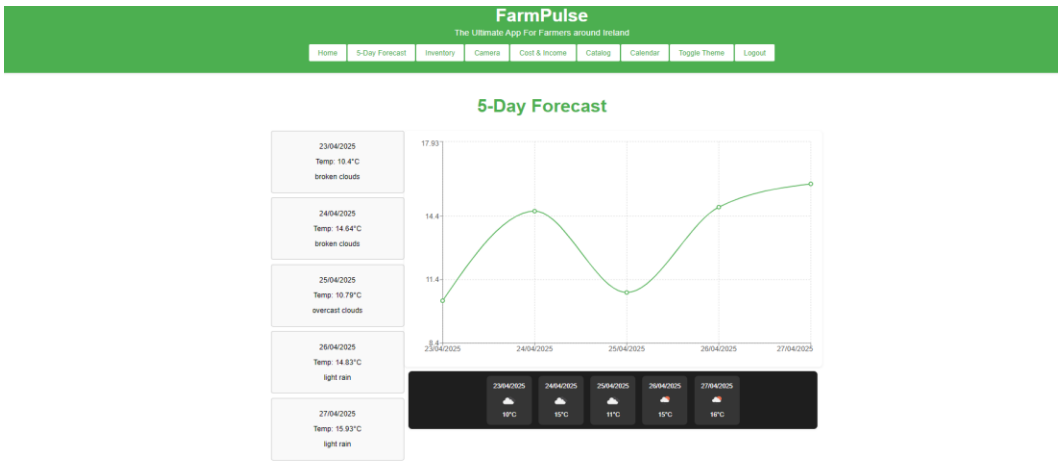
Figure 36: Home Page



Farm Inventory

Hay Bales	Bolus	Meal	lamblac
Bales	Tubs	Bags	Bags
Quantity: 16	Quantity: 1	Quantity: 5	Quantity: 5

Figure 37: Inventory Page



5-Day Forecast

23/04/2025
Temp: 10.4°C
broken clouds

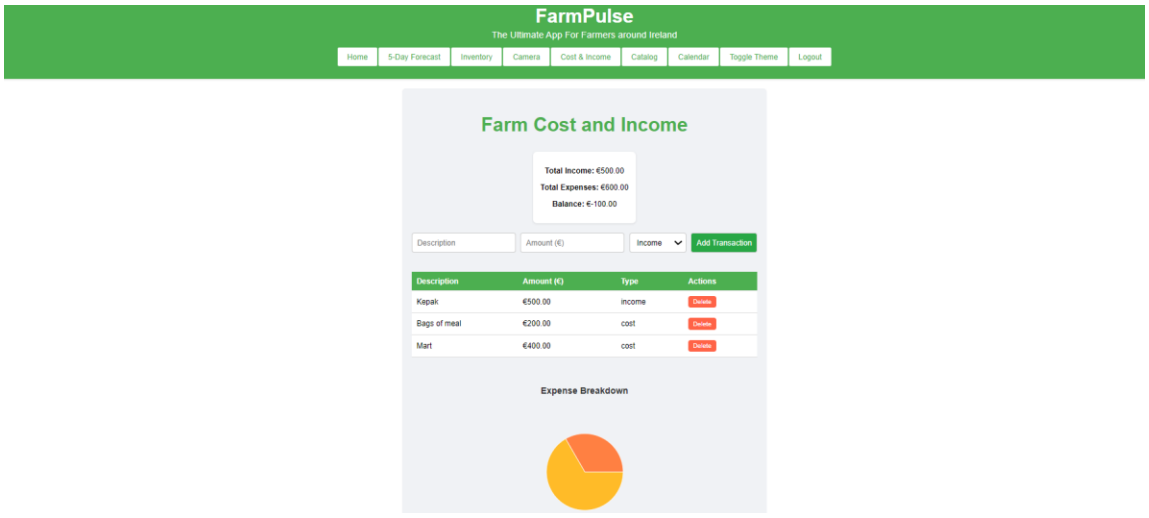
24/04/2025
Temp: 14.64°C
broken clouds

25/04/2025
Temp: 10.79°C
overcast clouds

26/04/2025
Temp: 14.83°C
light rain

27/04/2025
Temp: 15.93°C
light rain

Figure 38: Forecast Page



Farm Cost and Income

Total Income: €500.00
Total Expenses: €600.00
Balance: €-100.00

Description	Amount (€)	Type	Actions
Kepak	€500.00	income	Delete
Bags of meal	€200.00	cost	Delete
Mart	€400.00	cost	Delete

Expense Breakdown

Figure 39: Cost Income Page

14 Appendix

The link to my GitHub repo can be found below:

https://github.com/G00293495/FarmingApp

15 References

[1] Google, " Google Maps Platform Documentation". [Online]. Available : https://developers.google.com/maps/documentation

[2] OpenWeather, "OpenWeatherMapAPI". [Online]. Available: https://openweathermap.org/

[3] MongoDB, "Atlas Database" [Online]. Available: https://www.mongodb.com/products/platform/atlas-database

[4] Open AI, "ChatGPT". [Online]. Available : https://chatgpt.com/ [Accessed - March 2025]

[5] BrightData, "Serp API". [Online]. Available: https://docs.brightdata.com/scraping-automation/serp-api/introduction

[6] Axios, "Get Started". [Online]. Available: https://axios-http.com/docs/intro

[7] React, "React". [Online]. Available : https://react.dev/

[8] NPM, "Multer". [Online]. Available: https://www.npmjs.com/package/multer

[9] NPM, "bcryptjs.react". [Online]. Available: https://www.npmjs.com/package/bcryptjs-react

[10] <Recharts />, "Recharts Guide". [Online]. Available: https://recharts.org/en-US/guide

[11] Node.js. [Online]. Available : https://nodejs.org/en

[12] Teagasc, “Challenges and Indicators for Resilient and Sustainable Farming systems”. [Online]. Available: https://www.teagasc.ie/news--events/daily/environment/challenges-and-indicators-for-resilient-and-sustainable-farming-systems.php

[13] Altassian, “Get Started With jira- A comprehensive Beginners Guide”. [Online]. Available: https://www.atlassian.com/software/jira/guides/getting-started/introduction