Smart Agricultural Application For Promoting Natural Fertilization

by

Fatma Tuğçe Terzi

Engineering Project Report

Yeditepe University

Faculty of Engineering

Department of Computer Engineering

2025

Smart Agricultural Application For Promoting Natural Fertilization

APPROVED BY:

Prof. Dr. Mert Özkaya . . . . . .
(Supervisor)

Prof.Dr. Şebnem Baydere . . . . . .

Assist. Prof. Dr. Onur Demir . . . . . .

DATE OF APPROVAL: 16/06/2025

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my advisor, Prof. Dr. Mert Özkaya, for his invaluable guidance, constructive feedback, and unwavering support throughout the development of this project. His profound knowledge in the field of intelligent systems, along with the perspectives I gained from the Intelligent Systems Workshop he conducted, has been a deep source of inspiration in shaping the core ideas of this work.

I would also like to extend my sincere appreciation to Fatih Çolak, whom I consulted during this project. His expertise in the field of agriculture provided critical insights that significantly contributed to the development of the smart agricultural application system. Furthermore, I am deeply grateful to Fatih Çolak for his dedicated efforts in raising awareness about organic farming in Turkey, which have played a pivotal role in shaping public discourse. His insightful contributions and the knowledge he shared with me have been invaluable to this endeavor.

I would also like to convey my heartfelt thanks to my parents for their unconditional love, patience, and encouragement throughout my academic journey. Their steadfast support has been my foundation.

Finally, I am thankful to my friends for their constant motivation and constructive feedback. Their companionship has not only made this journey possible but also immensely rewarding.

ABSTRACT

Smart Agricultural Application For Promoting Natural Fertilization

This study introduces an integrated decision-support platform that enables small- and medium-scale farmers to plan, monitor, and optimise sustainable crop production. At its core is a crop-rotation algorithm that pairs soil texture classes with nitrogen-fixing species and successive crop families, thereby minimising synthetic fertiliser use while maximising long-term soil fertility. A complementary organic-protection module provides evidence-based biological control prescriptions for insects, diseases, weeds, nematodes, and rodents, along with synergistic intercropping suggestions for growers seeking diversified planting patterns.

The mobile front end—developed with React Native and Expo Router—offers agenda-style panels in which users can design three- or five-year rotation calendars and record biological control activities for future reference. Farmers enter field geometry, real-time GPS coordinates (via GSM), soil type, current crop, and sensor IDs; live weather data are fetched automatically, while nutrient values are entered manually. NestJS-based backend services combine these inputs to generate time-series soil-nutrient graphs and fertiliser recommendations tailored to each crop–soil profile.

An RSS-driven news feed, sourced from official Turkish agricultural outlets, delivers policy updates and best practices without informational clutter. By merging precision-agriculture tools with regenerative approaches, the system offers an accessible, data-driven environment that encourages long-term vision, reduces chemical inputs, and enhances farm profitability.

ÖZET

TARLAYOLDASI

TarlaYoldası, çiftçilerin tarımsal süreçlerini daha verimli, bilinçli ve veri odaklı şekilde yönetmelerini sağlamak amacıyla geliştirilen web tabanlı bir tarım yönetim mobil platformudur. Kullanıcı dostu arayüzü ve esnek mimarisi sayesinde tarımsal verilerin toplanması, analiz edilmesi ve anlamlı bilgilere dönüştürülmesini hedefler. Hem web hemde mobil arayüzü mevcuttur.

Ekin Rotasyonu, Yan Yana Ekim ve Organik Koruma sayfalarıyla toprağın azot dengesini optimize eden rotasyon dizileri, sinerjik ekin eşleştirmeleri ve böcek-hastalık-yabancı ot gibi zararlılar için biyolojik mücadele reçeteleri sunar. Plan ve Ajanda panelleri, organik koruma geçmiş kayıtlarını (ör. böcek, hastalık, yabancı ot müdahaleleri) ile çiftçinin bizzat oluşturduğu ekin rotasyon takvimlerini birbirinden ayrı alanlarda saklar. Böylece kullanıcı hem geçmiş biyolojik mücadele uygulamalarını geriye dönük inceleyebilir hem de üç-beş yıllık geleceğe yönelik rotasyon planlarını güvenle kaydedip tek ekranda yönetebilir. Tarlalarım arayüzünde kullanıcı GSM kullanarak gerçek konum bilgilerini ve o konuma özel hava durum bilgilerini görür. O tarlaya özel sensör ekleme sayfası açılır. Sensörler de manuel olarak girilmiş toprak besin değerlerini ve bunların tarihe göre grafiksel değişimini görür. Kullanıcı manuel olarak atanmış toprak besin değerlerini gördükten sonra dikmek istediği ekin türünü seçtikran sonra o ekin türüne özel veritabanından büyüme evreleri gelir ve kullanıcı bunu da seçtikten sonra en uygun gübreler sıralanır ve gübre detayları seçilen kartlarda çıkar.O gübrenin kullanım koşulları, yöntemi ,dozajı gibi. Ek olarak, resmi Türk tarım kaynaklarından beslenen Haber Modülü güncel mevzuat ve en iyi uygulamaları filtreleyerek paylaşır; Forum alanı ise çiftçiler arasında bilgi alışverişini teşvik eder. Bu bileşenlerin tümü, uygun maliyetli sensörlerin yaygınlaşmasıyla birlikte otomatik veri toplamayı da destekleyecek biçimde ölçeklenebilir tasarlanmıştır. Böylece TarlaYoldası, hassas tarım ile rejeneratif yaklaşımları harmanlayarak kimyasal girdileri azaltan, uzun vadeli vizyonu teşvik eden ve çiftlik kârlılığını artıran kapsamlı, veri odaklı bir ekosistem sunar. Uygun fiyatlı toprak sensör teknolojilerinin eksikliği nedeniyle, sistem şu anda gerçek zamanlı sensör entegrasyonunu simüle etmek için manuel veri girişine dayanmaktadır. Ancak, aynı sistem mimarisi gelecekte IoT sensörlerinin entegrasyonunu destekleyerek, gerçek sensörler kullanılabilir hale geldiğinde sorunsuz bir adaptasyon sağlayacaktır. Uygulanan yöntemler, gelecekte sensör tabanlı veri toplama için esneklik sağlayacak şekilde tasarlanmıştır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS/ABBREVIATIONS

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
CRUD	Create, Read, Update, Delete
DI	Dependency Injection
DOM	Document Object Model
DTO	Data Transfer Object
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
JWT	JSON Web Token
JSON	JavaScript Object Notation
MQTT	Message Queuing Telemetry Transport
NPK	Nitrogen, Phosphorus, Potassium
ORM	Object–Relational Mapping
REST	Representational State Transfer
SQL	Structured Query Language
TS	TypeScript
UI	User Interface
UX	User Experience
SQLite	Structured Query Language Lite

1. INTRODUCTION

1.1. Overview

Agriculture has historically been one of the most strategic sectors due to its role in meeting humanity's fundamental nutritional needs. However, in contemporary agricultural production, the widespread use of chemical fertilizers, pesticides, and agrochemicals aimed at increasing yield has emerged as a long-term threat to both environmental and human health (Aktar, Sengupta & Chowdhury, 2009). The World Health Organization (WHO) and the Food and Agriculture Organization of the United Nations (FAO) report that the intensive use of such chemicals degrades the biological structure of soil, pollutes groundwater resources, and lowers the quality of agricultural products (FAO, 2021). In the specific context of Türkiye, the dominance of conventional agricultural practices underscores the pressing need for development in the areas of organic farming, sustainable production, and digitalization.

In light of these developments, sustainable agricultural policies and organic farming practices are gaining increasing importance. While organic farming offers an environmentally friendly, economically sustainable, and socially equitable production model, its adoption by farmers remains limited due to barriers such as lack of access to information and insufficient technical expertise (Scialabba & Hattam, 2002). Moreover, beginner farmers often encounter technical difficulties in production processes, limited access to traditional knowledge sources, and a lack of adequate consultancy services, which collectively hinder the widespread adoption of organic farming.

In this context, the smart agriculture mobile application developed within the scope of this project aims to provide a multifunctional platform that facilitates the transition to organic farming, enhances farmers' access to information, and integrates digital agricultural practices into the field. The application consists of three core modules:

TarlaYoldası is a web-based farm-management mobile platform that unifies agronomic planning, data collection, and decision support in a single, modular ecosystem. The front end—built with React Native and Expo Router—offers intuitive dashboards where farmers design multi-year crop-rotation calendars, log past biological control measures, and visualise field data through map-based and chart-based views. A NestJS back end provides RESTful services for authentication, soil-nutrient analytics, and a rules engine that aligns soil texture, nitrogen-fixing species, and fertiliser recommendations. In its current "sensor-simulation" mode, the system generates mock readings for moisture, pH, and macronutrient levels when users register virtual sensor

IDs; the layered architecture is, however, pre-configured for real-time IoT integration as soon as affordable probes become available. Complementary modules—such as organic pest management, companion-plant matching, RSS-driven agricultural news, and a farmer forum—promote knowledge sharing and regenerative practices. Together, these components create a scalable, data-driven environment that enhances productivity, reduces chemical inputs, and empowers small- and medium-scale farmers to make timely, evidence-based decisions.

1.1.1. IoT Sensor Usage and Future Integration

The current release of TarlaYoldası operates in a sensor-simulation mode because low-cost, field-ready IoT soil probes are not yet widely available. Farmers can register a sensor ID to receive simulated readings for key soil parameters—moisture, pH, and macronutrient levels—allowing them to familiarise themselves with data-driven workflows while hardware is still out of reach. The platform's layered architecture has been engineered for seamless migration to real devices: once compatible probes are deployed, live measurements will stream into the system in real time, eliminating manual input and dramatically improving the precision and timeliness of agronomic recommendations.

1.1.2. Requirements

The requirements for the TarlaYoldası system can be categorized into functional and non-functional requirements:

1.1.3. Functional Requirements

This section enumerates all functional requirements of the *TarlaYoldaşı* system. These are organized by module, including user management and security, field management, sensor systems, organic fertilizer and pest control, crop rotation planning, and API services.

1. User Management and Security Users must be able to register, authenticate, and maintain secure sessions. The system shall support:

- **Registration**: Capture first name, last name, email address, and password to create a new user account.
- **Login**: Allow existing users to sign in using their email and password.
- **Session Management**: Issue and validate JWTs to maintain authenticated sessions.

- **Endpoint Protection**: Restrict access to protected API routes, ensuring only authenticated users may invoke them.

2. Field Management and Weather Integration The system shall enable farmers to register and administer field parcels, and retrieve environmental data for each field.

- **Add Field**: Record a new field's name, area, geographic location, soil type, and intended crop type.
- **Geolocation**: Obtain GPS coordinates for each field and display it on a map.
- **List Fields**: Present a summary of all registered fields in a paginated or filterable list.

Weather Integration

- **Retrieve Weather**: Fetch real-time temperature, humidity, and wind speed data for each field location.
- **Display Weather**: Present current weather conditions in the field detail view.

3. Sensor System The sensor subsystem must support adding, associating, and reading multiple sensor data types.

- **Add Sensor**: Register a new sensor by its unique identifier.
- **Associate Sensor**: Link a sensor device to a specific field.
- **Update Metadata**: Modify sensor properties such as name or location.

4. Organic Fertilizer Management Farmers must have access to a catalog of organic fertilizers and receive tailored recommendations.

- **List Fertilizers**: Display available organic fertilizer types.
- **Fertilizer Details**: Provide application methods, recommended dosages, and storage instructions for each type.

- **Generate Recommendations**: Analyze sensor data to suggest suitable fertilizers.
- **Tailored Suggestions**: Adjust recommendations based on crop type and growth stage.
- **Rule-Based Logic**: Apply predefined rules to generate consistent recommendations.

5. Organic Pest Control The system shall allow tracking of both chemical and biological pest control methods.

- **Record Treatment**: Log each pest type encountered and the chosen control method.
- **Agent Recommendations**: Suggest biological agents appropriate for each pest.
- **Application Guidance**: Provide dosage and application instructions.

6. Crop Rotation To sustain soil health, the system shall facilitate planning and tracking of crop rotations.

- **Create Plans**: Enable users to define multi-year rotation schedules.
- **Soil-Based Recommendations**: Suggest rotations optimized for the field's soil type.
- **History Maintenance**: Store past rotation plans for reference.

- **Combination Suggestions**: Recommend beneficial crop pairings for intercropping.

7. News Farmers require timely updates on agricultural news.

Agricultural News

- **Fetch News**: Retrieve relevant articles via RSS feeds.
- **Filter Topics**: Display only agriculture-related items.
- **Real-Time Updates**: Push new headlines as they arrive.
- **Source Linking**: Provide links back to the original news articles.

8. Communication System The system shall support community interaction through forum messaging.

- **Forum Posts**: Allow users to create and categorize discussion threads.
- **Message History**: Display chronological conversation records in each thread.

10. API Management A robust API layer is essential for integrations and automation.

- **Endpoint Design**: Define RESTful routes for all core resources.
- **Documentation**: Generate up-to-date API documentation (e.g., via Swagger).
- **Error Handling & Logging**: Implement consistent error responses and log all API activity.

1.1.4. Non-Functional Requirements

Security Requirements

- The system shall store sensitive user data using cryptographic hashing.
- The system shall enforce access control so that users can only access their own data.
- Input validation shall be enforced on all DTOs using `class-validator`.
- Validation errors shall be returned as structured responses via `ValidationPipe`.
- Only predefined endpoints shall be exposed; all controllers shall define explicit routes.

Data Management Requirements

- The system shall cache weather data for 24 hours by field (stored in `weather` field).

Integration & Data Accuracy Requirements

- The system shall integrate with the OpenWeatherMap API to fetch real-time weather data.
- The system shall provide location-based weather data tailored to each field's coordinates.
- The system shall display weather data with an accuracy of ±1 °C.

Compatibility & Maintainability & Architectural Requirements

- The system shall use a single codebase via React Native for both Android and iOS.
- The system shall run on iOS 12.0 and above.
- The system shall run on Android 8.0 and above.
- The system shall use a modular architecture with over 20 distinct modules (e.g., users, fields, sensors).

Performance Requirements

- The system's average API response time shall be below 500 ms.
- Under high CPU load, 95% of requests shall respond within 1 second.
- The 95th percentile (P95) response time shall not exceed 1 second.
- Under a load of 100 simultaneous requests, the system's average response time shall be below 1 second.
- The system shall use optimized native components on each platform to ensure native performance.

Memory Requirements

- Continuous request streams shall not exhibit memory leaks.
- Under sustained load, memory usage shall remain below 100 MB.

Usability & Responsive Design Requirements

- The system shall employ a responsive design to adapt the UI to various screen sizes.

Reliability Requirements

- Data integrity and consistency shall be ensured via TypeORM transaction management and entity validations.
- Error handling shall use precise exceptions (e.g., `NotFoundException`, `ConflictException`) for invalid parameters or conflicts.
- HTTP responses shall be structured with appropriate status codes (2xx for success, 4xx for client errors, 5xx for server errors).

Backend Infrastructure Requirements

- Node.js shall serve as the event-driven, non-blocking I/O runtime.
- NestJS (v11.1.3) shall provide a TypeScript-based modular framework.
- TypeScript shall enforce static typing across the codebase.
- Express.js shall operate as the underlying HTTP server.
- SQLite shall be used as the relational database engine.
- TypeORM (v0.3.24) shall implement the repository-pattern ORM for data access.
- ESLint shall perform static code analysis.
- Prettier shall enforce consistent code formatting.

1.2. Problem Statement

One of the principal challenges in contemporary agricultural practice is the effective management and analysis of agronomic data. Current systems often disperse critical components—such as soil nutrient profiles, crop type information, fertilizer recommendations, and farmer-to-farmer communication—across disparate and fragmented platforms. This fragmentation impairs producers' ability to make informed, timely decisions, thereby undermining both productivity and sustainability objectives. Moreover, the lack of robust knowledge-sharing mechanisms within

farming communities hampers the dissemination of best practices and the transfer of valuable experiential insights.

In response to these issues, the proposed project aims to develop an integrated mobile and web-based platform expressly designed to address these challenges. The platform will consolidate functionalities including crop-specific growth-stage monitoring, an intelligent organic fertilizer recommendation system, a forum-based information-exchange environment, and the delivery of up-to-date agricultural news via RSS feeds. It will also incorporate advanced analytics for data-driven decision support, weather integration for location-based recommendations, and comprehensive crop management tools.

By doing so, the system will empower farmers with data-driven decision-making tools, optimize the use of resources, and facilitate the broader adoption of sustainable farming practices through an accessible, integrated interface that bridges traditional agronomic knowledge and modern technological capabilities.

1.3. Objectives

Develop a Rule-Based Organic Fertilizer Recommendation Module Implement a decision-support engine that processes soil analysis data (pH, N, P, K, moisture, temperature, EC), crop type, and growth stage to generate tailored organic fertilizer recommendations, including fertilizer type, dosage, application frequency, and timing.

Implement an Analytical Crop Rotation Planning System Create a data-driven module that formulates optimal crop rotation schedules to preserve soil health and enhance long-term productivity by analyzing historical planting records, soil profiles, and crop compatibility rules.

Design a Biological Pest Management Recommendation Engine Construct a pest□control system that suggests intercropping combinations and biological control agents to reduce reliance on chemical pesticides, detailing expected efficacy, application methods, and maintenance guidelines.

Enhance Digital Accessibility in Rural Areas Develop a user-friendly mobile interface for iOS and Android—featuring simplified navigation, large icons, and local-language support—to accommodate users with limited digital literacy and improve rural farmers' access to the platform.

Establish a Forum and News Module for Knowledge Sharing Build a category- and tag-based discussion forum where farmers can exchange experiences and local success stories, alongside an RSS-driven news feed covering organic agriculture policies, market opportunities, and research updates to boost user engagement.

Construct a Modular, Scalable Architecture for Future Integration Design a decoupled backend using NestJS that facilitates seamless integration of IoT sensor data and future enhancements (e.g., AI-based analytics), ensuring maintainability and extensibility.

Integrate Location-Based Weather Services Provide real-time, field-specific weather summaries (including wind speed, humidity, and precipitation forecasts) by integrating geospatial weather APIs, thereby enabling farmers to optimize the timing of agricultural activities.

1.4. Scope

This project encompasses the design, implementation, and preliminary performance evaluation of a smart-agriculture platform consisting of a cross-platform mobile application—developed with React Native and Expo—and a NestJS-based TypeScript REST API. Within the study, manually entered soil-analysis data are processed to deliver rule-based modules for organic-fertilizer recommendations, crop-rotation planning, biological pest management, and intercropping strategies, all supported by a SQLite database and JWT-based authentication. The scope further includes the integration of location-specific weather data via the OpenWeatherMap API and the deployment of a hierarchical forum alongside an RSS-driven agricultural news feed. Real-time IoT sensor acquisition, advanced machine-learning algorithms, and an independent web interface lie outside the current scope and are reserved for future work.

2. Background

2.1. Previous Works in Smart-Agriculture Systems

Smart-farming research has progressed from isolated decision-support tools to integrated mobile ecosystems that combine sensor networks, cloud analytics, and farmer-centric interfaces. Early platforms such as *AgriApp* and *FarmLogs* offered real-time crop management, weather forecasting, and resource-use optimization, setting a precedent for precision agriculture in countries like the Netherlands (Zhang, Wang Wang, 2002; van der Meer et al., 2019). Collaborative solutions—including the *Connected Growers Network* and the *AgroSense* project—extend these capabilities by enabling growers to share granular field data and by monitoring soil moisture and temperature to fine-tune irrigation and fertilisation (Jansen de Groot, 2021).

Recent systematic reviews highlight the diversity and impact of mobile applications such as *Plantix* and *Kisan Suvidha*, which support traceability and disease diagnosis (Doe & Roe, 2023). Similarly, Kumar et al. (2022) report significant knowledge gains among Haryana farmers who rely on smartphones for agricultural guidance. Nevertheless, meta-analyses reveal persistent adoption gaps in regions dominated by smallholders—Türkiye included (Korkmaz et al., 2023).

2.2. Evolution of Sensor Technologies

Advances in low-cost soil-sensor hardware—ranging from capacitive moisture probes to solid-state pH and EC electrodes—have converged with long-range, low-power IoT networks such as LoRaWAN and NB-IoT. This convergence allows parcel-level data acquisition with multi-year battery life, enabling near-real-time monitoring of key soil parameters that drive fertiliser and irrigation decisions (Smith Lee, 2024).

2.3. Data Analytics and Decision Support

While cloud-based machine learning services power most large-scale platforms, recent studies demonstrate the feasibility of edge-level inference on mobile devices for tasks such as nutrient-deficiency diagnosis and rule-based fertiliser scheduling. Lightweight analytical models reduce latency and mitigate unreliable rural connectivity, making decision support accessible even in bandwidth-constrained areas.

2.4. Regulations and Incentives in Türkiye

Türkiye's Ministry of Agriculture and Forestry has expanded digital farming incentives through programs such as TARBL and the *e-Tarm Portal*, and recent updates to organic farming certification guidelines emphasize site-specific nutrient management. However, current national platforms focus on macroscale meteorological data and lack parcel-level, crop-specific guidance.

2.5. Limitations of Existing Integrated Applications

Field studies show that 62 % Turkish farmers obtain agricultural information primarily via smartphones, yet they must juggle multiple single-purpose apps for soil analysis, market prices and community advice (Yılmaz Öztürk, 2022). This fragmentation hampers adoption and increases cognitive load, especially among users with limited digital literacy.

2.6. Identified Research Gap

Taken together, previous work confirms the transformative potential of mobile smart farming tools but also exposes a critical gap: the absence of an all-in-one, Turkish-language mobile solution that (i) integrates real-time soil sensor analytics, (ii) delivers rule-based organic fertilizer and crop rotation guidance attuned to local agronomic conditions, and (iii) fosters community knowledge exchange through news and forum modules. This thesis addresses that gap by designing and evaluating a modular, cross-platform application tailored to the socio-economic and agro-ecological context of Türkiye.

2.7. Related Work

Previous studies on smart agriculture systems consistently highlight the benefits of mobile and sensor-driven platforms for increasing productivity, optimizing input use, and achieving sustainability goals. Early solutions such as *AgriApp* and *FarmLogs*zhang2002, vanderMeer2019 illustrate how real-time crop management, weather intelligence, and resource use analytics streamline on-farm decision-making. Collaborative initiatives like the *Connected Growers Network* and the *AgroSense* project extend this paradigm by facilitating data sharing among producers and automating irrigation and fertilization through continuous soil moisture and temperature monitoringjansen2021. Comprehensive reviews further underscore the diversity of mobile tools from disease diagnosis applications such as *Plantix* to traceability services like *Kisan Suvidha*—and their positive impact on knowledge transfer and the adoption of best practices-doe2023, kumar2022.

Despite these advances, technology uptake remains uneven in regions dominated by small and medium-sized farms; in Türkiye, limited digital literacy, heterogeneous soil properties, and local climatic constraints diminish the applicability of foreign-developed systems. Consequently, the literature evidences both the promise of sensor-based, data-driven farming and the critical need for an integrated, Turkish□language mobile solution that (i) delivers localized soil analytics, (ii) offers rule-based organic fertilizer guidance, and (iii) embeds community communication features—thereby overcoming app fragmentation that currently impedes widespread technological adoption in the Turkish agricultural sector.

3. ANALYSIS DESIGN

3.1. Introduction

This system is developed using modern web technologies and features a scalable and modular architecture. With its user-friendly interface, secure data management, and infrastructure prepared for future integration of IoT-based real-time monitoring, it enables efficient and centralized management of agricultural processes. This section provides an in-depth analysis of the methodologies adopted for the system's development and explains in detail the architectural design, the technologies used, and the implementation strategies that ensure efficiency and reliability.

3.2. Analysis

3.2.1. Use Case Diagram

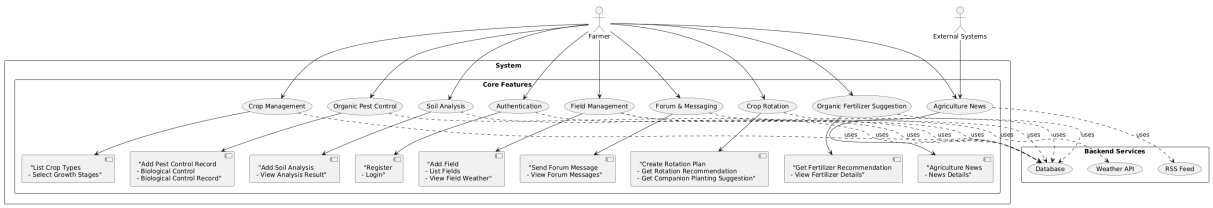


Figure 3.1: Use Case Diagram

Purpose: Allows users to register and log into the system.

User Actions:

- Registers or logs in upon first opening the application.

Relations with Other Use Cases:

- Prerequisite for all other modules.

Purpose: Enables users to manage their agricultural fields and view associated data such as

location and weather.

User Actions:

- Adds new fields.
- Views a list of existing fields and their details.
- Accesses current weather data for each field.

Relations with Other Use Cases:

- Directly linked to Soil Analysis, Crop Management, and Crop Rotation modules.
- Required before these modules can be used.
- Weather view feature is integrated with a Weather API.

Purpose: Allows users to upload and review soil analysis data for specific fields.

User Actions:

- Records soil analysis data.
- Reviews results for decision making.

Relations with Other Use Cases:

- Dependent on Field Management.
- Feeds data into Organic Fertilizer Suggestion and Crop Management modules.

Purpose: Suggests appropriate organic fertilizers based on soil and crop data.

User Actions:

- Receives organic fertilizer recommendations.

- Views historical records of applied fertilizers.

Relations with Other Use Cases:

- Connected to Soil Analysis and Field Management.
- Fertilizer suggestions depend on soil condition, crop type, and growth stage.

Purpose: Enables selection of crop types and their growth stages.

User Actions:

- Selects the crop type.
- Specifies the crop's growth stage.

Relations with Other Use Cases:

- Used in fertilizer suggestion logic.

Purpose: Assists in planning crop rotation based on soil type to improve productivity.

User Actions:

- Creates and saves rotation plans.
- Views system-generated suggestions.
- Reviews intercropping recommendations.

Purpose: Provides a record and knowledge base for organic and biological pest control strategies.

User Actions:

- Records pest control activities.
- Views detailed instructions for organic methods.

Purpose: Offers current and filtered agricultural news content.

User Actions:

- Reads agricultural news.
- Accesses external sources via embedded URLs.

Purpose: Facilitates communication and knowledge sharing among users.

User Actions:

- Posts and reads messages in the forum.

Relations with Other Use Cases:

- Not technically dependent on other modules.
- Supports knowledge exchange within the community.

3.2.2. Sequence Diagram Login and Register

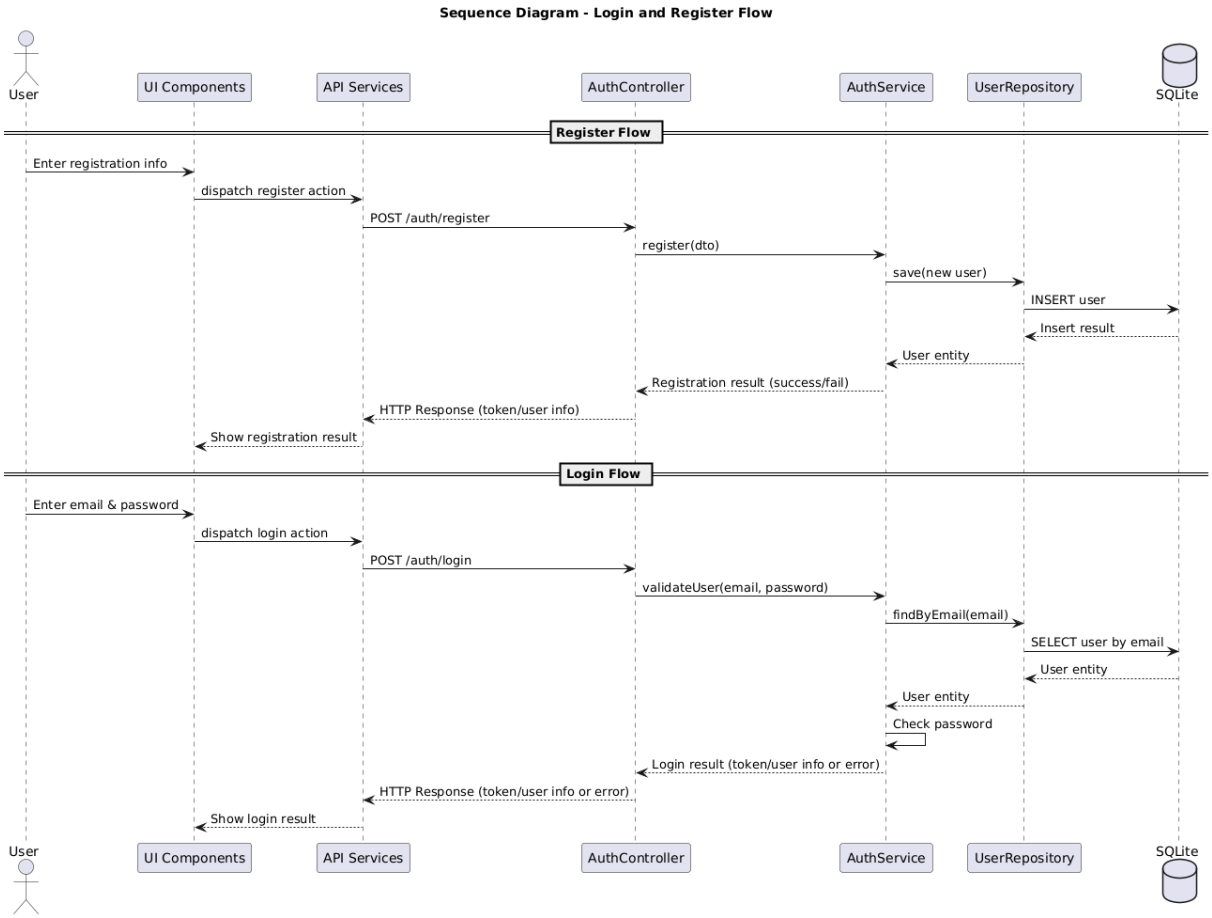


Figure 3.2: Sequence Diagram of the Login and Register Module

The sequence diagram depicts the user authentication workflow in a layered architecture, highlighting both registration and login processes. User actions in the UI trigger API requests, which are routed by the AuthController to the AuthService for business logic and validation. The AuthService interacts with the UserRepository to perform database operations (insert or query) in SQLite. Upon successful registration or login, the system returns a response (including a JWT token and user info) to the UI. This design ensures modularity, separation of concerns, and secure handling of user credentials.

3.2.3. Sequence Diagram - News Module system

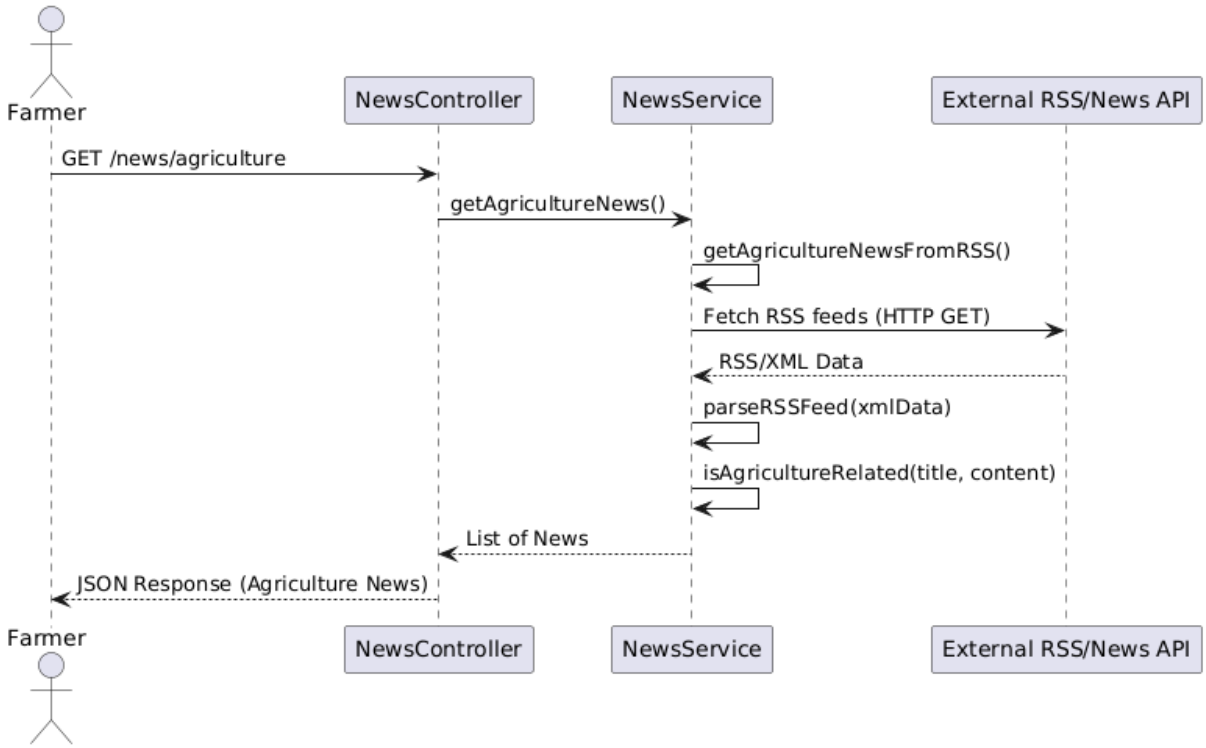


Figure 3.3: Enter Caption

When a user (farmer) wishes to access agricultural news through the application interface, they initiate a request to the relevant API endpoint (e.g., /news/agriculture). Upon receiving this request, the NewsController handles the incoming call and delegates the task to the getAgricultureNews() method of the NewsService. The NewsService is responsible for retrieving news articles from external sources by invoking its getAgricultureNewsFromRSS() function. This function systematically sends HTTP GET requests to multiple agricultural news RSS feeds (External RSS). The returned RSS/XML data is then parsed within the service, where only news items pertinent to agriculture are filtered and selected. Ultimately, the curated list of agricultural news articles is delivered to the user as a JSON response via the NewsController.

3.2.4. Sequence Diagram Field Management

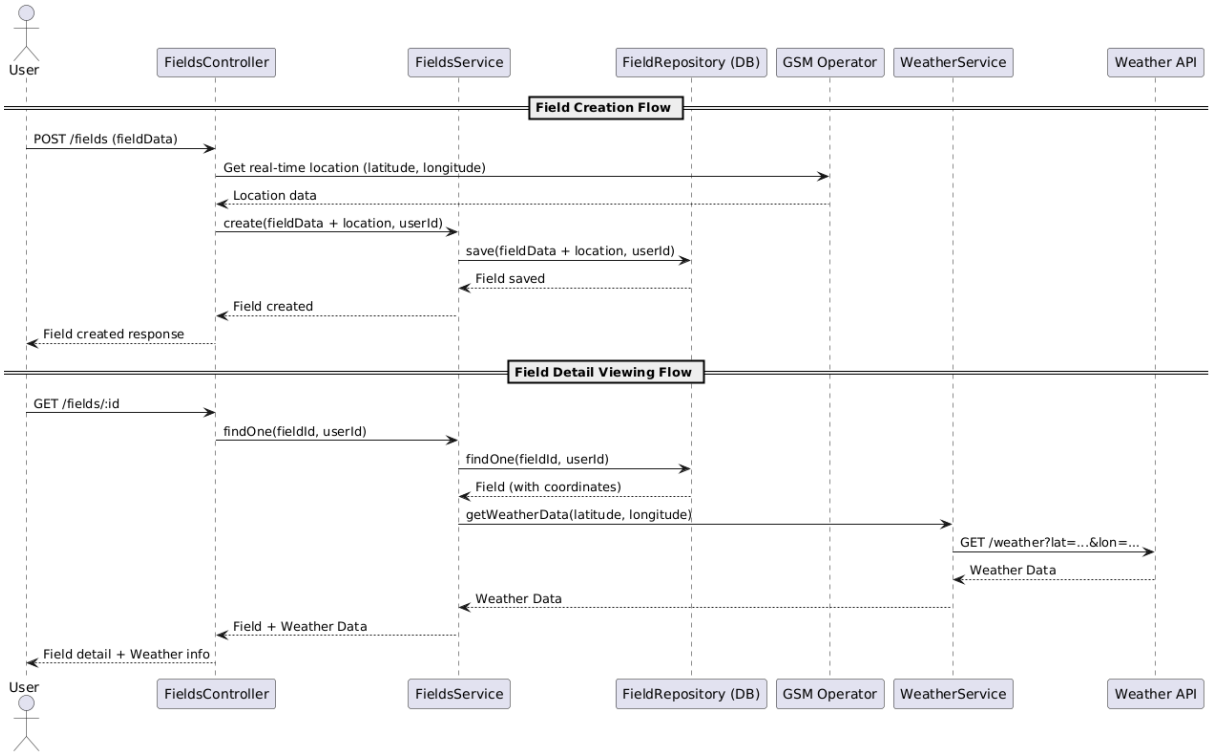


Figure 3.4: Field Management Sequence Diagram

Field Creation Flow

User Initiates Field Creation: The process starts when the user submits an HTTP `POST` request to the `/fields` endpoint, including the new field's metadata (e.g., name, area, crop type).

Retrieving Real-Time Location: Upon receiving the request, the `FieldsController` queries the GSM Operator's location service to obtain the user's current latitude and longitude, ensuring that the field is registered with precise coordinates.

Location Data Integration: The GSM Operator returns the coordinates to the `FieldsController`, which then merges this geolocation data with the original field payload.

Field Data Persistence: The enriched payload is forwarded to `FieldsService` for business-logic processing. `FieldsService` invokes `FieldRepository.save()` to persist the new field record—together with its coordinates—into the database.

Confirmation and Response: Once the field is successfully stored, `FieldRepository` returns a confirmation to the `FieldsService`, which subsequently notifies the `FieldsController`. Finally, the controller replies with an HTTP `201 Created` response, confirming that the field has been registered.

Field Detail Viewing Flow

User Requests Field Details: The client sends an HTTP `GET` request to `/fields/{id}` to retrieve detailed information about a specific field.

Fetching Field Data: `FieldsController` delegates the request to `FieldsService`, which queries `FieldRepository` to obtain the field record, including its stored coordinates.

Obtaining Weather Information: After retrieving the field data, `FieldsService` calls `WeatherService`, supplying the field's latitude and longitude. `WeatherService` forwards a request to an external Weather API for current meteorological conditions.

Weather Data Integration: The Weather API returns real-time weather data to `WeatherService`, which relays it to `FieldsService`.

Composing the Response: `FieldsService` merges the field record with the weather information to form a comprehensive DTO and returns it to `FieldsController`.

Delivering Information to the User: `FieldsController` serialises the DTO to JSON and responds with HTTP `200 OK`, enabling the client to display detailed field data together with up-to-date weather information.

3.2.5. Soil Analysis & Fertilizers Recommendations Sequence Diagram

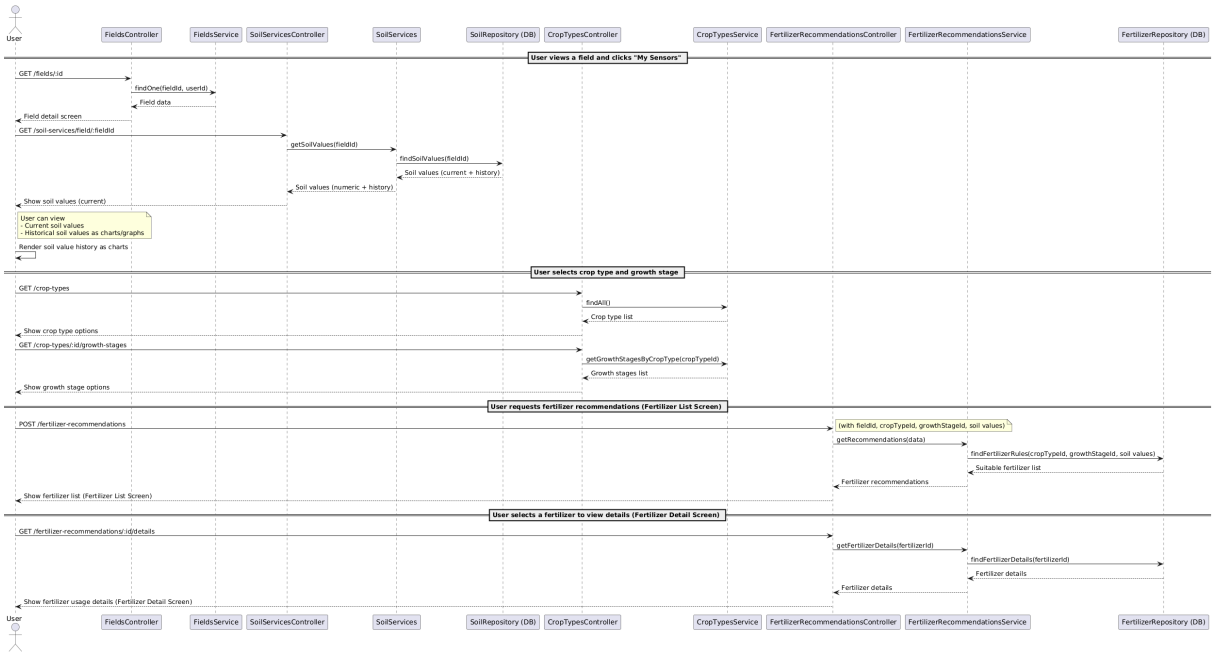


Figure 3.5: Fertilizers Recommendation sequence diagram

The sequence in Fig. 3.5 depicts the end-to-end workflow whereby a farmer (i) retrieves field details and soil-nutrient values, including historical trends; (ii) specifies the crop type and growth stage; and (iii) receives rule-based fertilizer recommendations with detailed usage guidelines. Each interaction reflects the project's layered architecture (*controller–service–repository*) and the corresponding REST endpoints.

Step 1: Viewing Field Details and Soil Values

The client issues `GET /fields/{id}`, handled by `FieldsController` (`ciftciyoldas-backend/src/fields/fields.controller.ts`). The controller delegates to `FieldsService`, which fetches the record from `FieldRepository`. Selecting *My Sensors* triggers `GET /soil-services/field/{fieldId}`, processed by `SoilServicesController` and its service implementation (`ciftciyoldas-backend/src/sensors/`). The service queries `SoilRepository` to return both current and historical nutrient measurements, enabling the UI to render interactive time-series charts.

Step 2: Crop Type and Growth Stage Selection

The farmer requests available crops via `GET /crop-types`, managed by `CropTypesController/Cro` (`ciftciyoldas-backend/src/crop-types/`). After a crop is chosen, the client retrieves its developmental phases with `GET /crop-types/{id}/growth-stages`, thereby specifying the current growth stage.

Step 3: Fertilizer Recommendation

The client submits a `POST /fertilizer-recommendations` request containing the field ID, crop type ID, growth stage ID, and latest soil values. `FertilizerRecommendationsControl` delegates to `FertilizerRecommendationsService` (`ciftciyoldas-backend/src/fertilizer-recommendations/`), which consults `FertilizerRepository` for matching rules and returns a curated list displayed on the *Fertilizer List Screen*.

Step 4: Retrieving Detailed Fertilizer Guidance

If the farmer selects a specific recommendation, the client calls `GET /fertilizer-recommendations/{id}/details`. The service retrieves the corresponding record from `FertilizerRepository` and responds with comprehensive application instructions.

Step 5: Visualising Historical Soil Data

When the user enters *My Sensors*, the backend returns a time-series of past measurements in addition to current values. The frontend leverages this dataset to generate dynamic charts, facilitating data-driven decisions on crop management and fertilizer application.

3.2.6. Organic Pest Control Module

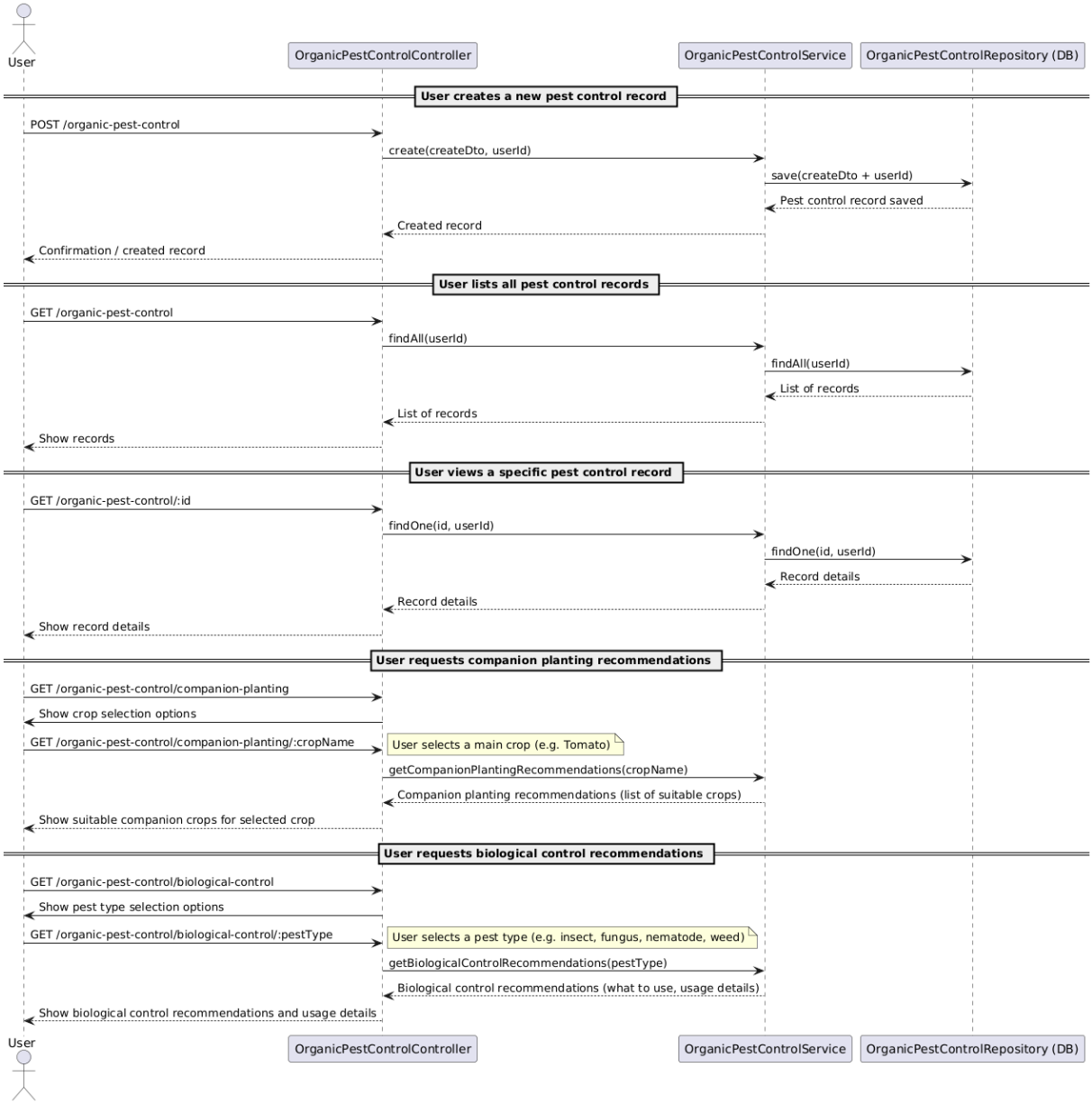


Figure 3.6: Sequence diagram for the Organic Pest Control module.

Figure 3.6 illustrates the interaction sequence for creating, listing, and retrieving organic pest–control records, as well as requesting companion–planting and biological-control recommendations. The workflow adheres to the project's layered architecture (*controller–service–repository*) and leverages the corresponding REST endpoints.

Step 1: Creating a New Pest Control Record

The user submits an HTTP `POST /organic-pest-control` request. `OrganicPestControlControl` receives the payload and forwards it to `OrganicPestControlService`. The service

processes the data and invokes `OrganicPestControlRepository.save()` to persist the record. Upon successful insertion, the repository returns a confirmation to the service, which propagates it back to the controller; the controller then responds with the confirmation or the newly created record.

Step 2: Listing All Pest Control Records

To retrieve every record, the client issues `GET /organic-pest-control`. The controller delegates the call to the service, which queries `OrganicPestControlRepository` for all records associated with the authenticated user. The resulting list flows back through the service to the controller and is presented to the user.

Step 3: Viewing a Specific Pest Control Record

To inspect a single entry, the client submits `GET /organic-pest-control/{id}`. The controller forwards the request to the service, which fetches the record from the repository using the supplied identifier and user context. The repository returns the entity; the service relays it to the controller, which delivers the detailed record to the user.

Step 4: Requesting Companion-Planting Recommendations

The user navigates to the crop-selection view via `GET /organic-pest-control/companion-planting`. After choosing a primary crop (e.g., tomato), the client invokes `GET /organic-pest-control/companion-planting/{cropName}`. The controller asks the service for suitable companion crops; the service returns the recommendation list, which the controller displays to the user.

Step 5: Requesting Biological-Control Recommendations

The user accesses the pest-type selector with `GET /organic-pest-control/biological-control`. After selecting a pest category (e.g., insect, fungus, nematode, weed), the client calls `GET /organic-pest-control/biological-control/{pestType}`. The controller consults the service, which retrieves pest-specific biological-control advice and usage details from the repository. The service forwards the recommendations to the controller, which presents them to the user.

3.2.7. Crop Rotation Plan Sequence Diagram

Figure 3.7 illustrates the interaction flow for creating, listing, and retrieving crop–rotation plans, as well as querying soil-specific recommendations and plan benefits. The workflow is

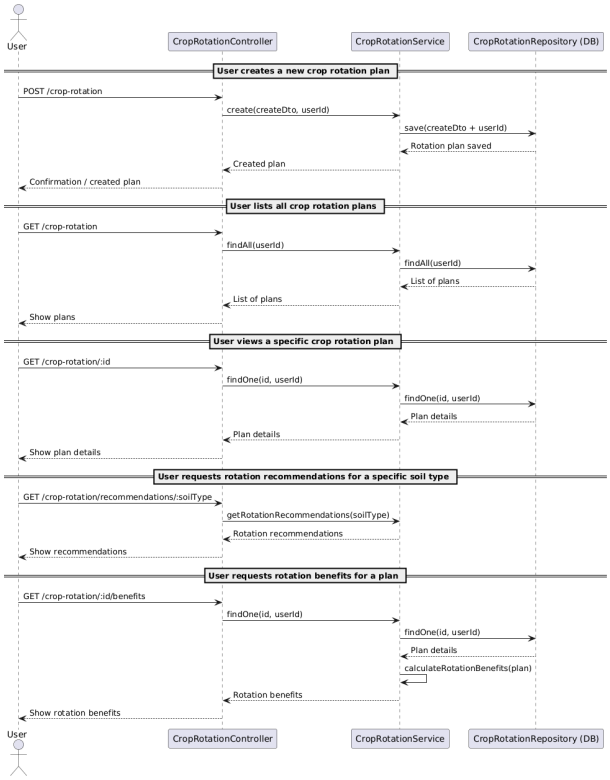


Figure 3.7: Crop Rotation Plan Sequence Diagram

implemented according to the layered backend architecture (*controller–service–repository*) and its corresponding REST endpoints.

Step 1: User Creates a New Crop-Rotation Plan

The client issues an HTTP `POST /crop-rotation` request. `CropRotationController` receives the payload and forwards the *createDto* together with the user identifier to `CropRotationService`. The service persists the plan via `CropRotationRepository.save()`. Upon success, the repository returns the newly created entity to the service, which in turn relays it to the controller; the controller responds with either a confirmation or the plan details.

Step 2: User Lists All Crop-Rotation Plans

To obtain every plan, the client submits `GET /crop-rotation`. The controller delegates the call to the service, which queries `CropRotationRepository` for all plans associated with the authenticated user. The resulting collection flows back through the service to the controller and is presented to the user.

Step 3: User Views a Specific Crop-Rotation Plan

The client requests `GET /crop-rotation/{id}`. The controller forwards the identi-

fier and user context to the service, which retrieves the plan from the repository. The repository returns the entity; the service passes it to the controller, which then displays the plan details.

Step 4: User Requests Rotation Recommendations for a Soil Type

The client calls `GET /crop-rotation/recommendations/{soilType}`. The controller relays the request to the service, which derives soil-specific rotation suggestions (algorithmic or rule-based) and returns them to the controller; the controller presents the recommendations to the user.

Step 5: User Requests Rotation Benefits for a Plan

To evaluate benefits, the client submits `GET /crop-rotation/{id}/benefits`. The controller first queries the plan via the service. `CropRotationService` retrieves the plan from the repository and executes `calculateRotationBenefits()`, returning the computed advantages to the controller, which forwards them to the user.

3.2.8. Forum Messages Module Sequence Diagram

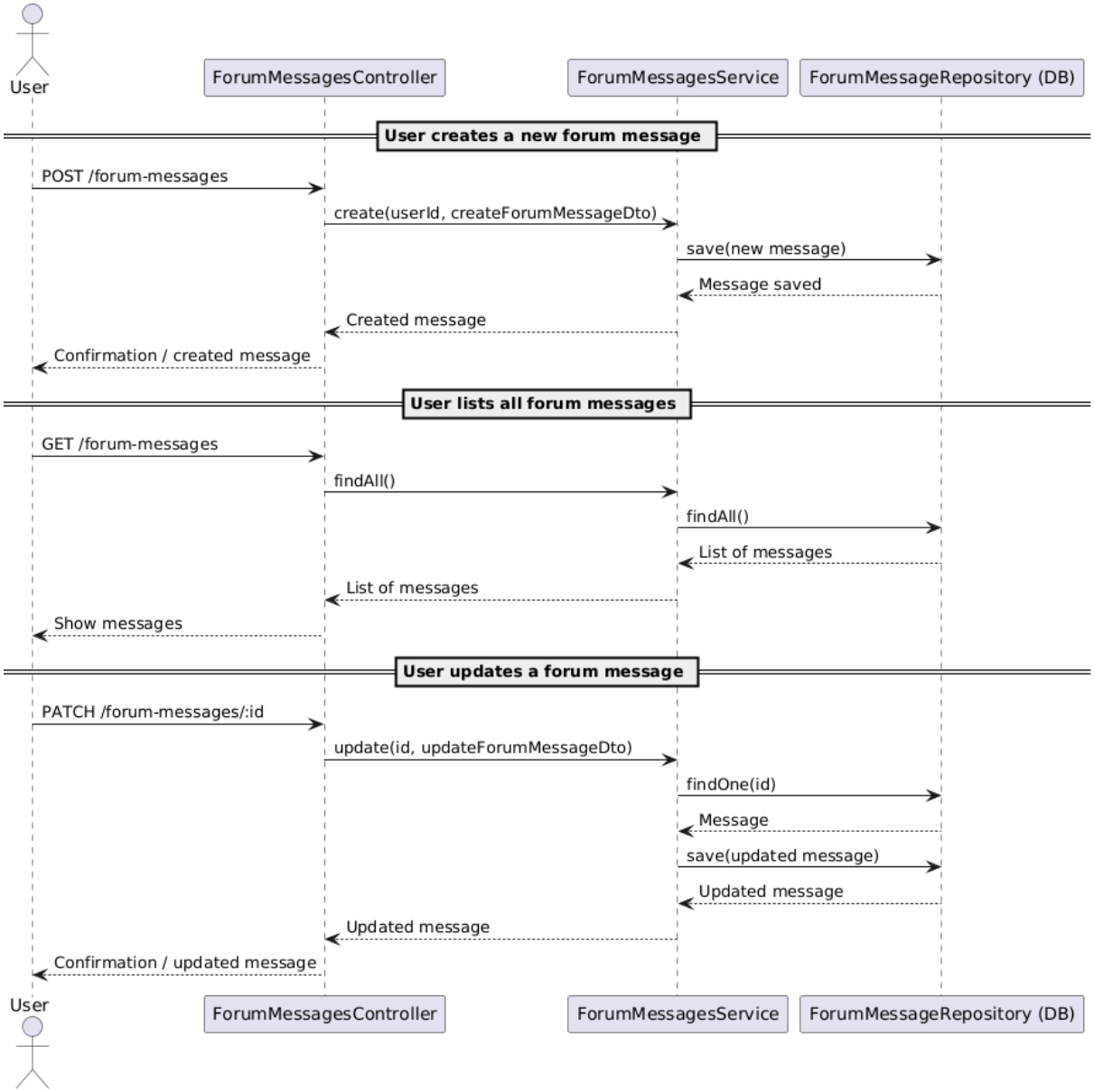


Figure 3.8: Forum Messages Sequence Diagram

Figure 3.8 shows the sequence of operations for creating, listing, and updating forum messages within the application. Each interaction follows the controller–service–repository pattern and utilises the designated REST endpoints.

Step 1: User Creates a New Forum Message

The client issues an HTTP `POST /forum-messages` request. `ForumMessagesController` receives the payload—including the *createForumMessageDto* and user identifier—and forwards it to `ForumMessagesService`. The service persists the record via `ForumMessageRepositor`

Upon success, the repository returns the created entity to the service, which relays it to the controller; the controller responds with a confirmation or the message details.

Step 2: User Lists All Forum Messages

To retrieve every message, the client submits `GET /forum-messages`. The controller delegates the request to the service, which invokes `ForumMessageRepository.findAll()`. The resulting collection flows back through the service to the controller and is displayed to the user.

Step 3: User Updates a Forum Message

To modify an existing entry, the client calls `PATCH /forum-messages/{id}`, supplying the *updateForumMessageDto*. The controller forwards the identifier, update data, and user context to the service. `ForumMessagesService` retrieves the target record via `findOne(id)`, applies the updates, and saves the entity back to the repository. The repository returns the updated message, which propagates through the service to the controller; the controller then returns a confirmation or the updated record to the user.

3.2.9. System Architecture and Component Design

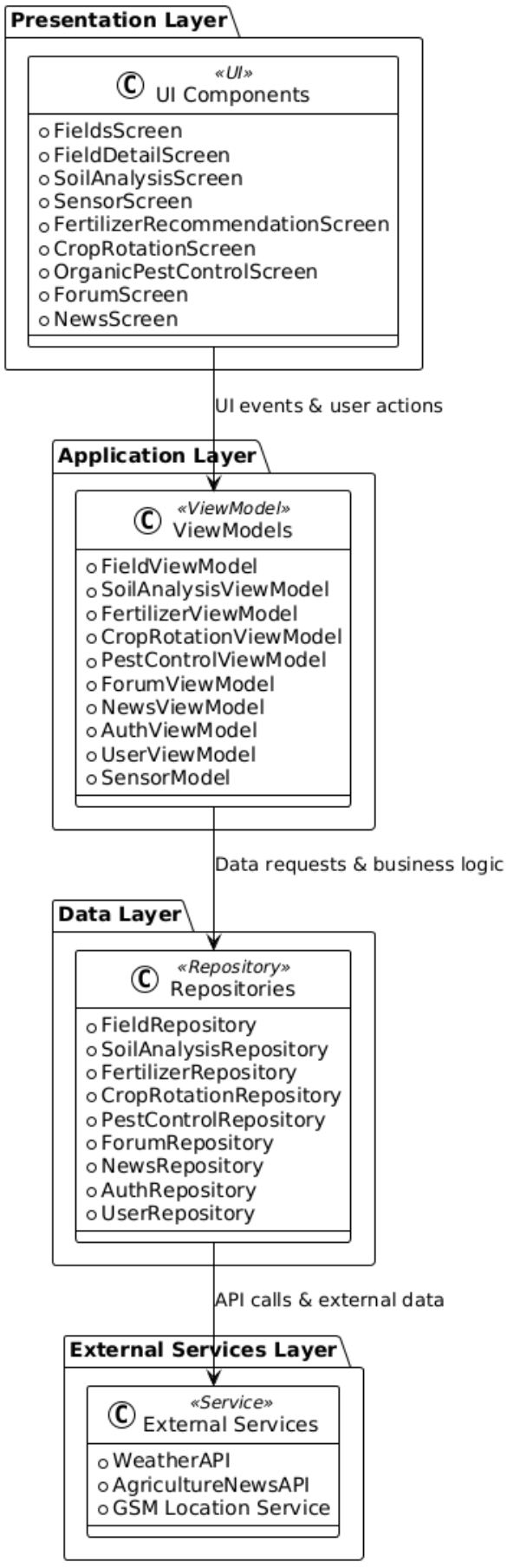


Figure 3.9: System Architecture

Figure 3.9 adopts a four-tier, layered architecture that enhances *sustainability*, *testability*, and *extensibility*. Each layer assumes a single, well-defined responsibility and communicates through a clear interface with its adjacent layer(s).

1. **Presentation Layer (UI)**

   Content: User-interface components and screens (e.g., *FieldsScreen*, *FieldDetailScreen*, *SoilAnalysisScreen*, *FertilizerRecommendationScreen*, *CropRotationScreen*, *OrganicPestControlScreen*, *ForumScreen*, *NewsScreen*).

   Role: Captures UI events and user actions, delegates them to the Application Layer, and renders results back to the user.

2. **Application Layer (ViewModels & Business Logic)**

   Content: ViewModel classes encapsulating state and domain logic (e.g., *FieldViewModel*, *SoilAnalysisViewModel*, *FertilizerViewModel*, *CropRotationViewModel*, *PestControlViewModel*, *ForumViewModel*, *NewsViewModel*, *AuthViewModel*, *UserViewModel*).

   Role: Processes user intentions received from the Presentation Layer, orchestrates business rules, and requests data from the Data Layer—thus acting as a bridge between UI and persistence.

3. **Data Layer (Repositories)**

   Content: Repository classes and data-access objects (e.g., *FieldRepository*, *SoilAnalysisRepository*, *FertilizerRepository*, *CropRotationRepository*, *PestControlRepository*, *ForumRepository*, *NewsRepository*, *AuthRepository*, *UserRepository*).

   Role: Fulfils data requests from the Application Layer, interacting with the database or external sources to fetch, store, and update persistent data.

4. **External Services Layer**

   Content: Third-party APIs and platform services (e.g., *WeatherAPI*, *AgricultureNewsAPI*, *GSM Location Service*, *Push*).

   Role: Executes outbound API calls initiated by the Data Layer and brokers communication between the application and external data providers.

Inter-Layer Flow

- *Presentation → Application:* User events are forwarded to the relevant ViewModel for processing.
- *Application → Data:* ViewModels invoke repository methods when data is required.
- *Data → External Services:* Repositories perform API calls whenever remote information is needed.
- *Reverse Flow:* Each layer returns its results upstream, culminating in visual feedback to the end-user.

3.2.10. Component Architecture

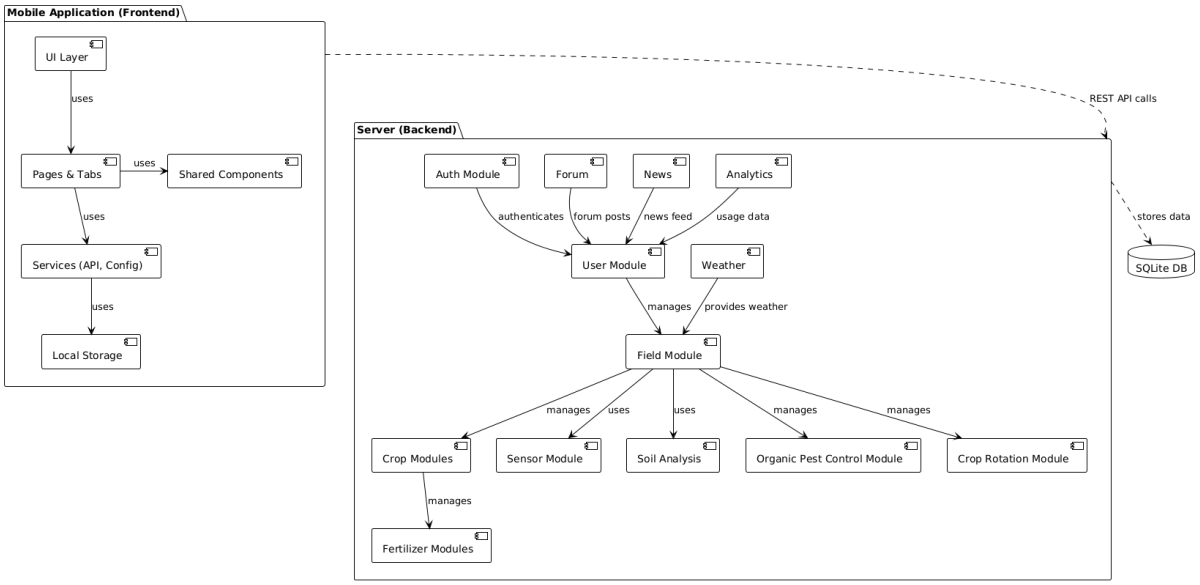


Figure 3.10: Component diagram

The diagram outlines the core architectural components of the *TarlaYoldaş* project and the relationships between these components. It is divided into two main sections: the Mobile Application (Frontend) and the Server (Backend). Additionally, data flow between modules and the centralized database is illustrated.

Mobile Application (Frontend)

The mobile application is developed using React Native and Expo, serving as the layer where users interact with the interface, view data, and perform actions.

- **UI Layer:** This layer manages the visual components of the application and facilitates user interaction.

- **Pages & Tabs:** These represent the primary screens and navigational sections of the application (e.g., Home, Fields, Chat).
- **Shared Components:** These are reusable visual or functional elements utilized across multiple pages (e.g., `ThemedText`, `Collapsible`).
- **Services (API, Config):** This sublayer handles communication with the backend, performs data transmission, and manages application configurations.
- **Local Storage:** Stores data on the user's device to support offline functionality and enhance access speed.

Relationships:

- The UI Layer interacts with Pages & Tabs.
- Pages utilize both Shared Components and Services.
- Services may read from or write to Local Storage when necessary.
- The entire frontend communicates with the backend via RESTful API calls.

Server (Backend)

The backend is developed using NestJS and is responsible for business logic, data management, and integration with external services.

- **Auth Module:** This module is responsible for managing user authentication and authorization processes. It implements a stateless authentication mechanism based on JSON Web Tokens (JWT) to ensure secure session management. The module utilizes Passport.js strategies to support both Local and JWT-based authentication, and incorporates bcrypt for secure password hashing. It features a robust token management system, issuing access tokens with a one-hour expiration and refresh tokens valid for seven days. Furthermore, the module provides route protection via `JwtAuthGuard`, ensuring that access to specific endpoints is restricted to authenticated users.
- **User Module:** Handles user registration and user-specific operations.
- **Field Module:** This module manages operations related to users' agricultural fields, including creation, updating, and analytical data collection. It integrates with a relational

database using the TypeORM entity structure and supports location-based field management through precise GPS coordinates (latitude/longitude). The module stores real-time weather data in JSON format via integration with the `WeatherModule`, and maintains associations with soil analysis records through well-defined `ManyToOne` and `OneToMany` relationships. It enables accurate area calculations (in decares) using decimal precision and enforces user-based data isolation to ensure secure and personalized data access.

- **Crop Modules:** Controls information regarding crop types, growth stages, and rotation planning.
- **Fertilizer Modules:** Tracks usage and suggestions for both organic and chemical fertilizers.
- **Sensor Module:** This module is integrated into the system architecture to facilitate the future connection of real soil sensors. Once a sensor capable of measuring soil nutrient levels (e.g., nitrogen, phosphorus, potassium) is developed, it can seamlessly transmit data to the system via an HTTP POST request to the `/sensors` endpoint. The sensor is matched with a predefined ID in the system, and the corresponding fields are automatically populated. Dedicated endpoints are also available for managing sensor connection status and user assignment. Owing to the flexible structure of the entity and DTO layers, the system can be easily extended to accommodate additional nutrient elements, and sensor data can be integrated both manually and automatically through IoT devices.
- **Soil Analysis Module:** This module is responsible for recording and evaluating the results of soil analyses. It stores key soil nutrient values (e.g., nitrogen, phosphorus, potassium) and tracks their historical changes over time. Additionally, it supports the visualization of these variations through graphical representations, enabling users to monitor soil health trends and make informed agricultural decisions.
- **Weather Module:** This module fetches location-specific weather information from external services and provides it to the agricultural fields. It integrates with the OpenWeatherMap API to retrieve real-time meteorological data, including temperature, humidity, wind speed, and precipitation status. Asynchronous data retrieval is performed using an HTTP client (`HttpService`) in combination with `RxJS`'s `firstValueFrom`, ensuring efficient and non-blocking operations. Environment variables such as the `OPENWEATHER_API_KEY` are securely managed via `ConfigService`, while data is localized using the metric unit system and Turkish language support. Type safety is ensured through the use of a strongly-typed `WeatherData` interface, and timestamp management is handled via Unix-to-Date conversions to maintain consistent temporal data representation.
- **Forum Module:** Enables open communication accessible to all users through forum messages, providing a shared space for public interaction and community exchange.

- **News Module:** This module delivers updates and news relevant to agriculture by integrating real-time data from multiple RSS feed sources, including the Ministry of Agriculture, agricultural news platforms, Google News, and Yahoo News, using the Axios HTTP client. It supports a variety of feed formats (RSS, Atom, and general XML) through a regex-based XML parser and performs keyword-based content classification using a domain-specific agricultural filtering algorithm. The module is integrated with a relational database via the TypeORM entity structure and stores news metadata—such as title, content, URL, image, publication date, and source—in JSON format. It ensures reliable data fetching through a 5-second timeout mechanism and custom User-Agent headers, and includes a fallback system that provides sample news content if RSS retrieval fails. Additionally, it supports content control through active/passive status flags and exposes RESTful API endpoints for full CRUD operations as well as customized agricultural news delivery.
- **Crop Rotation Module:** Sugges crop rotation plans for each field, allowing users to optimize planting sequences and improve soil health.
- **Organic Pest Control Module:** Records and manages organic pest control activities for enabling users to track biological and cultural pest management methods.

Relationships:

- The Auth Module relies on the User Module for identity verification.
- The User Module governs the Field Module.
- The Field Module directly interacts with the Crop, Sensor, and Soil Analysis Modules.
- Crop Modules are interconnected with Fertilizer Modules.
- The Weather Module provides meteorological data to the Field Module.
- Forum, News, and Analytics modules interface with the User Module.

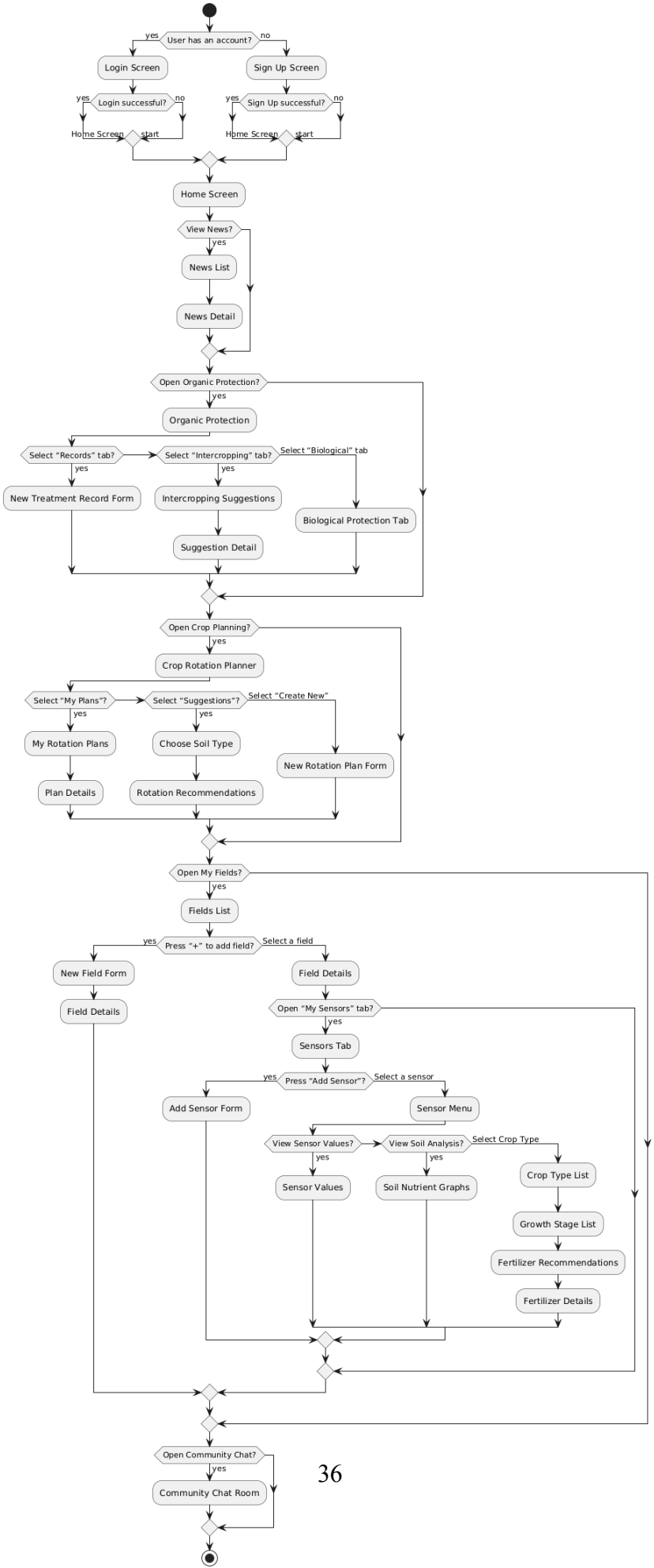
Database (SQLite)

All backend modules persist their data within a centralized SQLite database, ensuring secure and consistent management of both user and system data.

Cross-Layer Communication

- **Frontend–Backend Communication:** The mobile application communicates with the backend exclusively through REST API endpoints. User requests are sent to the backend, processed, and responses are returned to the frontend.
- **Backend–Database Communication:** Backend modules directly interact with the SQLite database to perform all read/write operations.

3.2.11. State Diagram



This diagram illustrates the transitions between screens and the fundamental functional structures of the developed mobile agricultural application. After completing the registration and login processes, users gain access to various modules via the home screen. The application provides digital support for a range of agricultural activities such as field management, crop rotation planning, fertilizer recommendation, biological control, and community interaction.

Main Modules:

- **CropRotation:** Enables users to plan crop rotations across different fields and crops. It includes screens for creating a new plan, receiving suggestions, and viewing plan details.
- **OrganicProtection:** A module for managing biological control strategies using natural methods. Users can view previous records or create new treatment entries.
- **CompanionSuggest:** Offers suggestions for companion planting based on the selected crop, helping users determine compatible plant pairings.
- **CommunityChat:** Provides an interactive environment where users can exchange ideas and communicate with each other via messaging.
- **FieldsList / CreateField:** Allows users to view existing fields and register new ones within the system.

Sensor System and Soil Analysis (SensorTab and Subscreens):

The screens located in the bottom-right section of the diagram represent the core components of the soil-analysis-based recommendation system. This structure is designed with the following features:

- **Dual-mode data input is supported:**
  - **Manual Entry:** Users can manually input sensor readings via the system interface.
  - **Automatic Entry (IoT Integration):** Real sensors (e.g., devices measuring nitrogen, phosphorus, and potassium levels in soil) can transmit data directly to the system using HTTP. The API infrastructure has been flexibly designed to support such device integrations.
- When sensors are registered in the system:

– The `AddSensor` screen is used to enter the sensor ID and register the device.

– The `SensorDetail` screen displays live data (`SensorValues`) and detailed soil analysis graphs (`SoilAnalysis`).

– The recorded soil nutrient values are associated with specific fields, enabling the fertilizer recommendation engine to function based on crop type and growth stage.

– Users can select the relevant crop and growth stage using the `CropTypePicker` and `GrowthStageList`, then receive personalized fertilizer suggestions via `FertilizerRecommendat`

Conclusion: The system is not limited to manual or test data. The developed infrastructure is designed to support automated data collection through real sensor device integration. As a result, real-time data collected from the field can support more accurate, timely, and recommendation-based agricultural decisions.

3.2.12. Class Diagram

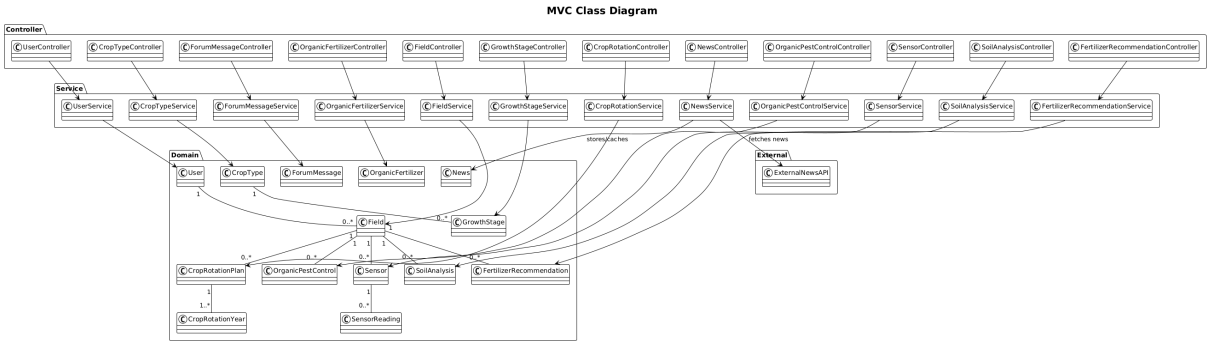


Figure 3.12: Class Diagram

3.2.13. Flow Chart

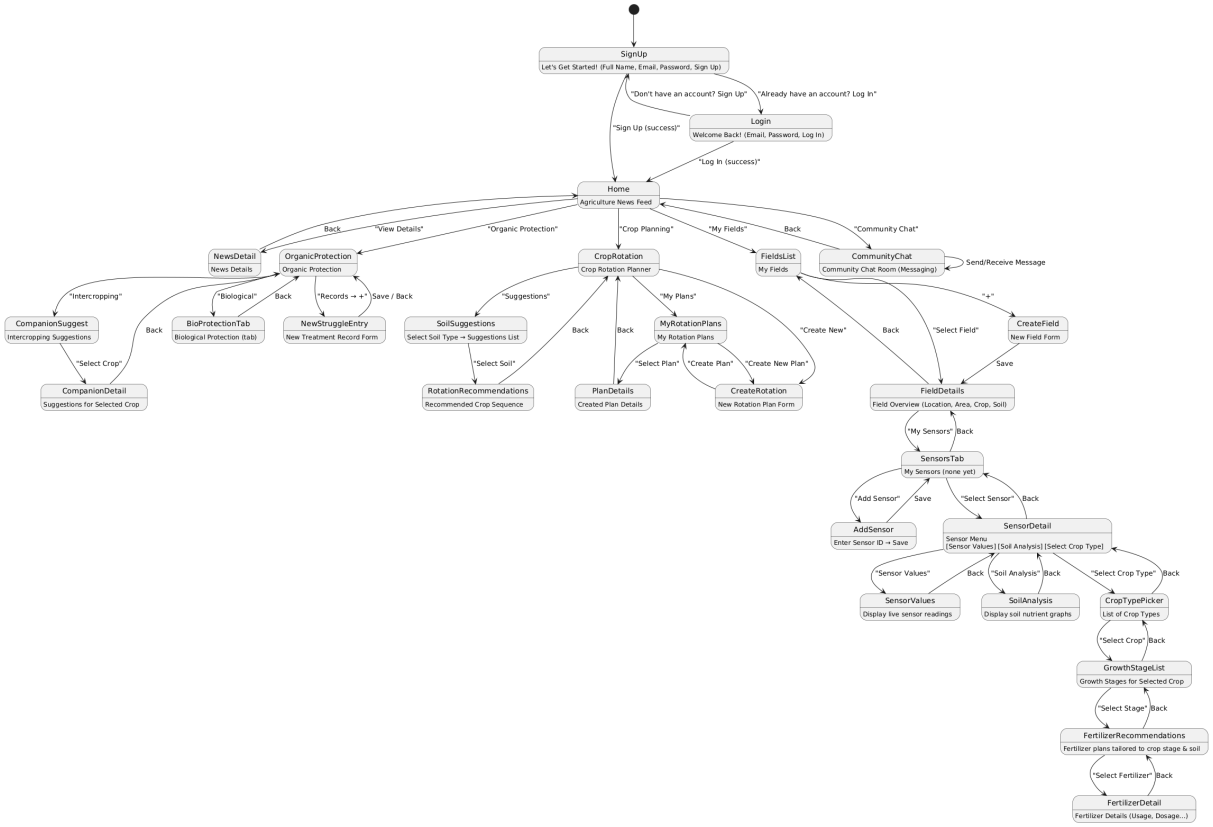


Figure 3.13: Flow Chart

3.2.14. ER Diagram

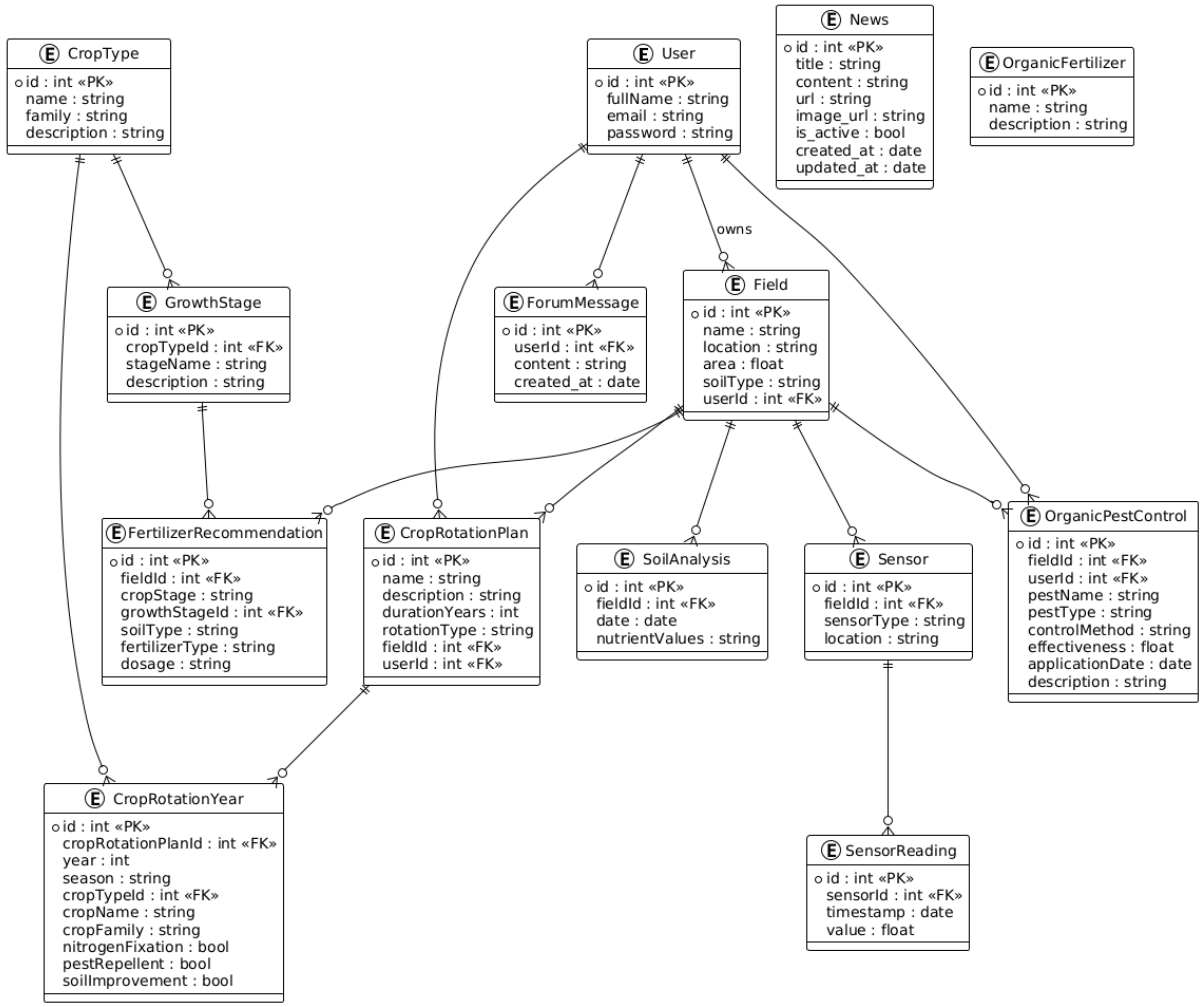


Figure 3.14: ER Diagram

3.2.15. Pseudocode

Authentication Module (auth.service.ts)

```
function login(email, password):
    user = UsersService.findByEmail(email)
    if user is null:
        return "Hatalı kullanıcı"
    if not bcrypt.compare(password, user.password):
        return "Hatalı şifre"
    token = JwtService.sign({ id: user.id, email: user.email })
    return token
```

```
function register(userData):
    if UsersService.findByEmail(userData.email) is not null:
        return "Email zaten kayıtlı"
    hashedPassword = bcrypt.hash(userData.password)
    user = UsersService.create({ ...userData, password: hashedPassword })
    return user

function refreshToken(refreshToken):
    decoded = JwtService.verify(refreshToken)
    if not decoded.isRefreshToken:
        return "Geçersiz refresh token"
    user = UsersService.findOne(decoded.sub)
    if user is null:
        return "Kullanıcı bulunamadı"
    newToken = JwtService.sign({ id: user.id, email: user.email })
    return newToken
```

Field Management Module (fields.service.ts)

```
function createField(fieldData, userId):
    field = new Field(fieldData)
    field.userId = userId
    save field to database
    return field

function getFields(userId):
    fields = find all fields where field.userId == userId
    for each field in fields:
        field.weather = WeatherService.getWeatherData(field.latitude, field.longitude
    return fields

function updateField(fieldId, updateData, userId):
    field = find field by id and userId
    if field is null:
        return "Tarla bulunamadı"
    update field with updateData
    save field
```

```
    return field
```

Soil Analysis Module (soil-analysis.service.ts)

```
function createSoilAnalysis(analysisData, fieldId):
    analysis = new SoilAnalysis(analysisData)
    analysis.fieldId = fieldId
    save analysis to database
    return analysis

function getSoilAnalyses(fieldId):
    analyses = find all analyses where analysis.fieldId == fieldId
    return analyses
```

Sensor Management Module (sensors.service.ts)

```
function createSensor(sensorData):
    if sensor with sensorData.sensor_id exists:
        return "Bu sensör zaten kayıtlı"
    sensor = new Sensor(sensorData)
    sensor.is_connected = false
    sensor.last_reading_at = now()
    save sensor
    return sensor

function connectSensor(sensorId, userId):
    sensor = find sensor by sensorId
    if sensor is null or sensor.is_connected:
        return "Bağlanılamadı"
    sensor.is_connected = true
    sensor.userId = userId
    save sensor
    return sensor

function updateSensorData(sensorId):
    sensor = find sensor by sensorId
```

```
    if sensor is null or not sensor.is_connected:
        return "Sensör bağlı değil"
    newData = generateSimulatedData()
    reading = new SensorReading(sensorId, newData, now())
    save reading
    update sensor with newData
    sensor.last_reading_at = now()
    save sensor
    return sensor

function getSensorHistory(sensorId):
    sensor = find sensor by sensorId
    if sensor is null:
        return []
    return sensor.readings
```

Crop Rotation Module (crop-rotation.service.ts)

```
function createCropRotationPlan(planData, userId):
    rotationSequence = []
    for dto in planData.rotation_sequence:
        rotationSequence.append(new CropRotationYear(dto))
    plan = new CropRotationPlan(planData)
    plan.rotation_sequence = rotationSequence
    plan.user_id = userId
    save plan
    return plan

function getCompanionPlantingSuggestions(cropName):
    companionPlants = {
        "Buğday": ["Fasulye", "Mercimek", "Nohut"],
        ...
    }
    return companionPlants[cropName] or []
```

Fertilizer Recommendation Module (fertilizer.service.ts)

```
function getRecommendedFertilizers(crop_type_id, growth_stage_id, nutrients):
    // Retrieve all fertilizer rules for the given crop type and growth stage
    rules = find all FertilizerRule where crop_type_id = crop_type_id and growth_stag

    matchedRules = []

    for rule in rules:
        // Map nutrient element names to internal field keys
        nutrientNameMapping = {
            'N': 'nitrogen_ratio',
            'P': 'phosphorus_ratio',
            'K': 'potassium_ratio',
            'Ca': 'calcium_ratio',
            'Mg': 'magnesium_ratio',
            'S': 'sulfur_ratio',
            'Fe': 'iron_ratio',
            'Zn': 'zinc_ratio',
            'B': 'boron_ratio'
        }

        nutrientKey = rule.nutrient_type
        mappedNutrientKey = nutrientNameMapping[nutrientKey]

        // Get nutrient value from sensor data
        value = nutrients[mappedNutrientKey] || nutrients[nutrientKey]

        // Check if value is valid
        if value is undefined or value is null or value is empty string:
            continue

        // Parse values as numbers
        numericValue = parseFloat(value)
        ruleValue = parseFloat(rule.value)

        // Compare based on operator
        if rule.operator == "<" and numericValue < ruleValue:
            matchedRules.append(rule)
        elif rule.operator == ">" and numericValue > ruleValue:
            matchedRules.append(rule)
```

```
        elif rule.operator == "<=" and numericValue <= ruleValue:
            matchedRules.append(rule)
        elif rule.operator == ">=" and numericValue >= ruleValue:
            matchedRules.append(rule)
        elif rule.operator == "=" and numericValue == ruleValue:
            matchedRules.append(rule)

    // Extract fertilizer info from matched rules
    recommendedFertilizers = []
    for rule in matchedRules:
        fertilizerInfo = {
            id: rule.fertilizer_id,
            name: rule.fertilizer.name,
            type: rule.fertilizer.type,
            description: rule.fertilizer.description,
            dosage: rule.dosage,
            application_method: rule.application_method,
            frequency: rule.frequency,
            recommended_amount: rule.recommended_amount,
            notes: rule.notes,
            nutrient_type: rule.nutrient_type,
            operator: rule.operator,
            threshold_value: rule.value
        }
        recommendedFertilizers.append(fertilizerInfo)

    return {
        rules: matchedRules,
        fertilizers: recommendedFertilizers,
        weatherAdvice: null
    }
```

Forum Module (forum.service.ts)

```
function createForumMessage(messageData, userId):
    message = new ForumMessage(messageData)
    message.userId = userId
    save message
```

```
    return message

function getForumMessages():
    return all ForumMessages

function updateForumMessage(messageId, updateData, userId):
    message = find message by messageId and userId
    if message is null:
        return "Message not found"
    update message with updateData
    save message
    return message
```

3.2.16. Architectural Overview

The TarlaYoldası is architected as a robust, modular, and scalable multi tiered platform, specifically tailored to address the unique challenges of modern agricultural management. The system leverages a layered architecture comprising a React Native + Expo frontend and a NestJS backend ensuring clear separation of concerns, maintainability, and extensibility. This design was selected to meet the demands of real time sensor data processing, offline usability, and seamless user experience across diverse devices and connectivity scenarios.

Data Reliability and Consistency The system employs persistent storage mechanisms (such as `AsyncStorage` on the frontend) and robust backend synchronization strategies to guarantee that critical agricultural data—such as forum messages and fertilizer recommendations—remains consistent and reliable. This is especially vital for users in rural or remote areas, where network connectivity may be intermittent. The backend ensures atomic operations and transactional integrity, while the frontend gracefully handles offline scenarios, queuing user actions and synchronizing them when connectivity is restored.

Scalability and Performance A strict separation between the presentation (UI), business logic (services), data access (repositories), and storage (database) layers allows each component to be scaled independently. For example, as the number of registered farmers or connected sensors increases, backend services and database resources can be scaled horizontally without impacting the frontend or other modules. The use of RESTful APIs and efficient data transfer formats ensures low-latency communication, even as the system grows.

Future Extensibility The architecture is designed with extensibility in mind. Abstraction layers and well-defined interfaces between modules (e.g., Forum, News, Fertilizer Recom-

mendation) allow for the seamless integration of new features or technologies. For instance, the system can easily accommodate additional sensor types, advanced AI based recommendation engines, or third party agricultural data sources, without requiring disruptive changes to the core architecture.

Security and Data Protection Security is enforced at every layer of the system. The backend utilizes JWT based authentication and role-based access control to ensure that only authorized users can access sensitive data or perform critical operations. Data validation is performed both on the client and server sides to prevent malicious input. All communication between the frontend and backend is encrypted, and sensitive user and agricultural data is stored securely in the database, adhering to best practices for data protection and privacy.

3.3. Methodology

The TarlaYoldası System is meticulously designed using a robust, scalable, and modular architecture that integrates real-time sensor data processing, cloud based backend services, and offline capable mobile applications. This chapter provides an in depth analysis of the methodologies adopted for the development of the system, detailing the architectural design, technologies utilized, and the implementation strategies that ensure efficiency and reliability.

3.3.1. Technologies Used

The TarlaYoldası System is built upon modern, cutting-edge technologies that enhance efficiency, scalability, and user experience. The following subsections detail the core technological components that enable the functionality of the system.

React Native and Expo

React Native, a powerful cross platform framework, combined with Expo, provides a streamlined development environment for building mobile applications. The user interface is developed using React Native's declarative UI components, ensuring a consistent and responsive user experience across different devices. The system implements reactive UI updates through state management libraries, ensuring seamless real-time responsiveness to data changes and user interactions. Navigation within the application is structured through React Navigation, providing consistent and predictable screen transitions while maintaining proper back stack management. The implementation follows a layered architecture pattern, utilizing services and repositories to efficiently manage data and maintain a clear separation of concerns between different applica-

tion layers.

The UI framework leverages React Native's component based architecture, enabling the development of modular UI components that enhance both code reusability and long term maintainability. This architectural approach significantly reduces boilerplate code, resulting in optimized UI rendering performance and improved overall application responsiveness.

NestJS Backend

NestJS serves as the backend framework, providing a structured and scalable architecture for handling server□side logic. The backend infrastructure supports RESTful API endpoints, facilitating real time data synchronization with minimal latency across diverse user populations. Security is implemented through JWT based authentication, ensuring robust user authentication protocols, while NestJS's built in validation and error handling mechanisms provide comprehensive protection for user data.

The system maintains functionality during periods of limited connectivity through sophisticated offline caching mechanisms, enabling continuous access to critical agricultural data. Integration with TypeORM enables efficient database operations, ensuring data integrity and consistency across the system.

AsyncStorage for Offline Persistence

AsyncStorage enables sophisticated data persistence capabilities directly on mobile devices, facilitating offline access to critical agricultural data. The system implements persistent storage mechanisms to ensure that user actions and data changes are queued and synchronized when connectivity is restored. This approach ensures that farmers can continue to access and manage their agricultural data even in scenarios with intermittent network connectivity.

4. IMPLEMENTATION

4.0.1. Introduction

The implementation of the TarlaYoldası represents a comprehensive approach that revolutionizes agricultural management through mobile technology. This innovative system integrates the NestJS backend framework, React Native mobile technology, TypeORM database management, and RESTful API design to create a robust tool tailored for modern farmers. The development process was not only focused on addressing current agricultural challenges but also aimed at establishing a foundation for future technological advancements in farming practices.

Primarily, the system was required to deliver accurate and timely agricultural recommendations based on complex environmental and soil data analysis. Secondly, maintaining high usability standards was essential to ensure that farmers with diverse technical backgrounds could effectively interact with the system's functionalities. Furthermore, the architecture was designed to ensure long-term viability and adaptability of the platform, currently supporting manual data entry while remaining fully compatible with future integration of IoT based soil sensors.

4.0.2. Presentation Layer (User Interface)

The presentation layer serves as the primary interface between farmers and the system's agricultural management capabilities. This layer has been implemented with careful attention to user experience principles specific to agricultural applications: Design Philosophy and Implementation: The user interface development followed a farmer centric approach, incorporating extensive feedback from agricultural professionals to ensure maximum utility and ease of use. The implementation focused on several critical aspects:

User-Centric Design: The interface is designed to be intuitive, with clear navigation and visual feedback. For example, the BuyumeScreen allows users to select growth stages for their crops, with each stage presented in a radio button format for easy selection. The DegerScreen displays soil analysis results in a grid layout, making it easy for users to interpret complex data at a glance.

Consistency: A uniform design language is maintained across all screens, including color schemes, typography, and button styles. The primary color palette includes shades of green which align with the agricultural theme of the application. Icons from the @expo/vector-icons library are used consistently to enhance visual cues.

Responsiveness: The UI adapts seamlessly to different screen sizes and orientations. Components like ScrollView and KeyboardAwareScrollView ensure that content remains accessible even on smaller screens or when the keyboard is active, as seen in the SignUp and SignIn screens.

Feedback and Validation: Forms include real time validation and error messages to guide users. For instance, the SignIn and SignUp screens validate email formats, password lengths, and confirm password matches, displaying errors immediately to prevent submission issues.

PRESENTATION LAYER WORKFLOW:

```
1. UI State Management
    MANAGE component state using useState hooks
    FETCH data from API using useEffect and fetch
    HANDLE loading states with ActivityIndicator
    STORE user data in AsyncStorage

2. User Interactions
    VALIDATE form inputs using basic validation
    PROCESS user actions through API calls
    UPDATE UI state based on API responses
    SHOW feedback using Alert.alert

3. Error Handling
    CATCH errors using try-catch blocks
    LOG errors using console.error
    SHOW user-friendly error messages
    PROVIDE retry options when possible

4. Data Display
    RENDER data using React Native components
    HANDLE empty states with placeholder UI
    SHOW loading indicators during API calls
    UPDATE UI when data changes
```

4.0.3. Application Layer (Business Logic)

The application layer represents the core business logic of the TarlaYoldası system, implementing rule based mechanisms to transform agricultural data into actionable insights. It acts as a critical bridge between user interactions and backend data processing while consolidating agricultural domain knowledge into a cohesive decision support system.

Multi-Parameter Analysis for Precision Agriculture:
The system integrates multi source data streams real time sensor inputs (e.g., soil moisture, temperature, nutrient concentrations), crop specific characteristics, and growth stage requirements to generate precise recommendations. By correlating these parameters, it identifies nutrient deficiencies and suboptimal growing conditions, supporting a comprehensive advisory framework.

Prioritized Recommendations:

- *Nutrient Deficiency Mitigation:* Evaluates soil composition (N–P–K ratios and micronutrients) and suggests tailored organic fertilizer blends with dosage guidelines.
- *Application Methods:* Recommends optimal delivery techniques such as foliar spray or soil incorporation depending on crop type and phenological stage.
- *Risk Assessment:* Flags adverse soil conditions (e.g., pH imbalance) and suggests remedial actions (e.g., lime treatment for acidic soils).

Dynamic Timing Optimization:
By tracking crop growth stages, the system determines optimal timing for fertilization, irrigation, and other agricultural interventions:

- *Vegetative Stage:* Emphasizes nitrogen input for foliage development.
- *Flowering Stage:* Suggests phosphorus and potassium supplementation to support blooming and fruit initiation.
- *Fruiting Stage:* Recommends balanced nutrient inputs with a focus on potassium to enhance fruit quality and yield.

Rule-Based Decision Making:
The system leverages agricultural domain rules and thresholds to assess soil conditions and crop specific needs:

- *pH Range Validation:* Ensures that soil pH aligns with optimal levels for specific crops (e.g., 6.0–7.5).
- *Nutrient Threshold Analysis:* Compares measured nutrient values against established agronomic baselines.
- *Growth Stage Correlation:* Maps current crop stage to relevant fertilizer strategies and application schedules.

Comprehensive Data Integration:

- *Historical Analysis:* Tracks past soil analysis reports and fertilizer application logs to detect usage trends and inform future recommendations.
- *Weather Integration:* Utilizes real time meteorological data to determine suitable timing and techniques for input application.
- *Crop Specific Logic:* Applies domain specific knowledge unique to each crop to improve recommendation accuracy.

Organic Farming Focus:

The system promotes sustainable practices by prioritizing organic farming methods:

- *Organic Fertilizer Suggestions:* Proposes natural alternatives including compost, vermicompost, and green manure.
- *Companion Planting:* Recommends beneficial plant pairings that support pest deterrence and soil enrichment.
- *Biological Pest Control:* Offers organic pest management strategies tailored to specific pest profiles and seasons.
- *Soil Health Monitoring:* Tracks improvements in soil structure and organic matter content over time.

Overall, the application layer ensures that farmers receive evidence based, practical, and ecologically responsible recommendations that enhance both productivity and soil sustainability.

```
1. Service Operations
    RECEIVE requests from controllers
    VALIDATE data using DTOs and class-validator
    PROCESS data through service methods
    APPLY business rules (fertilizer recommendations, crop rotation)
    RETURN results to controllers

2. Service Methods
    VALIDATE input using DTOs and class-validator
    PROCESS data through service methods
    APPLY business rules (fertilizer recommendations, crop rotation)
    UPDATE database through TypeORM repositories
    INTEGRATE weather data for recommendations

3. Error Handling
    LOG errors using console.error
    THROW appropriate exceptions (NotFoundException,
       UnauthorizedException, ConflictException)
    HANDLE errors through try-catch blocks
    RETURN error responses with HTTP status codes

4. Data Management
    HANDLE sensor data updates through TypeORM repositories
    VALIDATE data using DTOs and validation pipes
    MANAGE database operations through TypeORM transactions
    PROCESS agricultural data through service methods
```

4.0.4. Data Management Layer

The data management layer of the TarlaYoldası system is implemented as a modular architecture on an SQLite database using the TypeORM object relational mapping framework. This layer guarantees that all agricultural data are stored and managed securely, consistently, and efficiently.

Database Architecture and Entity Model. A comprehensive entity model encompasses every aspect of agricultural operations. Primary entities include `User` (user management), `Field` (field metadata), `SoilAnalysis` (soil testing results), `CropGrowth` (plant growth tracking), `OrganicFertilizerRecord` (organic fertilizer applications), `OrganicPestControl` (biological pest control measures), and `CropRotationPlan` (crop rotation schedules). Each entity defines domain specific attributes and inter entity relationships that mirror real world agronomic structures.

Relational Data Model. Data integrity is enforced through `ManyToOne` and `OneToMany` associations. For example, one `User` may own multiple `Field` records, and each `Field` can have numerous `SoilAnalysis`, `OrganicFertilizerRecord`, and `CropGrowth` entries. This hierarchical structure simplifies data retrieval and preserves referential integrity across related tables.

Data Validation and Security. Input validation is performed by DTO classes in conjunction with the `class-validator` library. Entity level constraints ensure agronomic accuracy for instance, soil pH values are restricted to the 0 14 range and nutrient quantities must be nonnegative preventing invalid data from being persisted.

Migration System. TypeORM's migration mechanism provides version control for the database schema. Each migration script applies or reverts a set of schema changes in a controlled manner. Existing migrations demonstrate the addition of sensor data tables, fertilizer application logs, biological pest control records, and the news module.

Data Access Layer. For each entity, a dedicated repository class implements the TypeORM Repository pattern to encapsulate all CRUD operations. This clear separation between data access logic and business logic enhances code reusability, testability, and maintainability.

Seed Data Management. A seed data framework populates development and test environments with essential reference datasets crop types, growth stages, organic pest□control presets, and crop rotation templates automating initial setup and ensuring consistent testing conditions.

Together, these components form a robust data management layer that underpins the TarlaYoldası system's scalability, reliability, and long term sustainability. "‘

```
1. DATA MANAGEMENT WORKFLOW:


    Monitor network connectivity (checkNetworkConnectivity)
```

```
    Maintain local cache freshness (maintainLocalCache)

    WHEN online:
        Sync with cloud (syncWithDatabase)
        Resolve data conflicts (resolveConflicts)
        Update local storage (updateLocalStorage)
        Notify UI/services of changes (notifyComponents)

2. Data Access Operations:

    FOR EACH data request:
        Check cache first (checkLocalCache)
            IF FRESH:
                Return cached data
            IF stale/missing:
                Update cache with new data
                Return latest data

3. Data Integrity Management:

    Validate data consistency (validateDataConsistency)
    Remove obsolete/corrupt data (cleanupObsoleteData)
    Optimize storage space (optimizeStorage)
    Backup critical data (backupCriticalData)
```

4.0.5. Implementation of the Fertilizer Recommendation System

The fertilizer recommendation system developed in this study was designed using the NestJS framework and TypeORM database management system. The system utilizes dependency injection through the FertilizerRecommendationsService, which provides a modular architecture for processing agricultural data and generating fertilizer recommendations. Database operations are handled through FertilizerRule and OrganicFertilizer repositories, while the data processing pipeline implements asynchronous programming using Promise-based operations. The system analyzes critical nutrient values including nitrogen (N), phosphorus (P), potassium (K), magnesium (Mg), boron (B), zinc (Zn), calcium (Ca), iron (Fe), and sulfur (S), evaluating them

against predefined threshold ranges.The rule evaluation mechanism uses basic filtering operations to compare sensor data with fertilizer rules. The system employs simple array operations and object mapping to process and structure data for frontend integration. TypeScript provides type safety through DTOs and class validator, while basic error handling manages null and undefined values. The modular architecture ensures maintainability and extensibility, allowing the system to adapt to future requirements. The fertilizer recommendation system provides a practical, data driven approach to supporting farmers in making informed agronomic decisions.

```
1. Service Initialization
    INJECT dependencies (FertilizerRule repository, WeatherService)
    CONFIGURE TypeORM repository for database operations
    SETUP basic error handling with try-catch blocks

2. Request Processing
    RECEIVE POST requests with DTO validation
    VALIDATE input parameters (cropTypeId, growthStageId, nutrients)
    FETCH fertilizer rules from database using TypeORM
    FILTER rules based on nutrient thresholds and operators
    RETURN matched rules with fertilizer details

3. Business Logic
    COMPARE sensor values against rule thresholds (<, >, =, <=, >=)
    APPLY weather conditions check (rainy, snowy, hot weather
       warnings)
    MAP nutrient types to sensor data keys (N, P, K, etc.)
    FILTER valid rules per crop type and growth stage

4. Error Handling
    LOG errors using console.error
    THROW appropriate exceptions (NotFoundException, etc.)
    HANDLE database query failures gracefully
    RETURN structured error responses

5. Data Management
    USE TypeORM repositories for database operations
    VALIDATE data using DTOs and class-validator
```

```
    MANAGE simple CRUD operations
    RETURN processed agricultural recommendations
```

4.0.6. Forum Messaging System Implementation

The forum messaging module provides a straightforward, HTTP based discussion system tailored for agricultural knowledge exchange. Messages are stored in a relational database with each entry linked to its author and, optionally, to a parent message for basic threading. Users may view and post messages via standard REST endpoints. While the backend supports message liking functionality through a dedicated endpoint, this feature is not currently implemented in the frontend interface. Data access is handled through TypeORM repositories, leveraging simple entity relationships without custom transaction controls or advanced loading strategies. Periodic polling (every ten seconds) retrieves new messages, and users may manually refresh the view to see the latest content.

```
FORUM MESSAGE OPERATION WORKFLOW:

1. Message Creation
    RECEIVE DTO with content and optional parent ID
    VALIDATE user authorization (JWT guard)
    ESTABLISH entity relationships (user association)
    PERSIST to database with timestamps
    RETURN complete message with user data

2. Message Retrieval
    QUERY database for all messages
    LOAD author relationships eagerly
    SORT by creation date (DESC)
    RETURN flat message list

3. Message Interaction
    IMPLEMENT simple like increments
    HANDLE basic CRUD operations
    USE standard TypeORM operations
    RETURN updated message data
```

```
4. Access Control
    ENFORCE JWT authentication on all endpoints
    USE basic user identification
    APPLY standard error handling
    MAINTAIN simple authorization checks
```

This implementation provides: Secure access control and data validation Efficient querying for both flat and threaded views Comprehensive error handling and logging Real time engagement tracking

4.0.7. News Module Implementation

The TarlaYoldası system's news module is designed to deliver up to date, agriculture specific information by integrating multiple RSS feeds with a fallback mechanism. This module provides farmers with a comprehensive news aggregation and filtering system to stay informed about sector developments.

RSS Feed Integration and Content Aggregation. Multiple parallel requests are issued via the Axios HTTP client (5 s timeout) to twenty agricultural news sources, including the Ministry of Agriculture and Forestry's official RSS feeds, specialized agribusiness sites (e.g., *tarimdan-haber.com*, *tarimhaberleri.com*), Google News (agriculture search), and Yahoo News RSS. Incoming XML from each feed—whether RSS 2.0, Atom, or generic XML is parsed using regex based routines to accommodate varied formats.

Content Filtering and Relevance Detection. A keyword-based filter identifies agriculture related articles by matching titles or body text against a bilingual list of over eighty terms. Keywords span crop species (wheat, maize, tomato), livestock (cattle, sheep, poultry), farm equipment (tractor, plough, combine harvester), and modern practices (organic farming, smart agriculture, precision farming). Only items containing at least one keyword are retained.

Fallback Mechanism. When RSS feeds fail to respond, the module automatically provides a set of seven exemplar articles covering topics such as organic farming, sustainable agriculture, smart farming technologies, seed breeding, irrigation systems, farmer support programs, and greenhouse cultivation. Aggregated items are sorted by date and limited to the most recent fifteen entries. Each article is assigned a placeholder image URL and undergoes link validation.

Frontend Integration and User Experience. The news feed is rendered in React Native using a card based layout. Each card displays the headline, summary, publication date, and source. Users may tap a card to open the full article via the Linking API. Loading states, error handling, and retry functionality optimize responsiveness, and a pull to refresh control enables manual updates.

Error Handling and Performance Optimization. RSS access errors are captured with `try-catch` blocks and logged per source. Feeds that fail to respond within five seconds are skipped. An early termination policy halts feed polling once ten valid articles have been collected. Requests include a browser like `User-Agent` header to mitigate scraping restrictions.

This implementation ensures that farmers can reliably and efficiently follow agricultural news while maintaining system stability and an optimized user experience.

```
1. RSS Feed Processing
    INITIALIZE multiple RSS feed URLs (20 sources)
    CONFIGURE axios HTTP client with 5-second timeout
    SET User-Agent header for web scraping compatibility
    ITERATE through feed URLs sequentially
    HANDLE individual feed errors with try-catch blocks

2. Content Parsing and Extraction
    PARSE XML data using regex patterns
    SUPPORT multiple RSS formats (RSS 2.0, Atom, general XML)
    EXTRACT title, description, link, and publication date
    CLEAN HTML tags from content using regex replacement
    VALIDATE extracted data completeness

3. Agriculture Content Filtering
    APPLY keyword-based filtering system
    CHECK title and content against 80+ agriculture keywords
    INCLUDE Turkish and English agriculture terminology
    REQUIRE minimum 1 agriculture keyword match
    LOG filtering results for debugging

4. Content Management and Fallback
```

```
    SORT news by publication date (DESC)
    LIMIT results to maximum 15 news items
    VALIDATE and sanitize URLs
    GENERATE placeholder images for missing visuals
    PROVIDE sample agriculture news as fallback

5. Error Handling and Performance
    LOG feed access errors individually
    SKIP feeds that exceed 5-second timeout
    TERMINATE early when 10 news items collected
    RETURN fallback content when RSS fails
    MAINTAIN system stability through error isolation

6. Frontend Integration
    DISPLAY news in card-based layout
    SHOW title, content preview, date, and source
    ENABLE click-to-open external URLs
    PROVIDE loading states and error messages
    IMPLEMENT pull-to-refresh functionality
    HANDLE URL validation before opening links
```

4.0.8. Field Module Implementation

The TarlaYoldası system's field management module provides a comprehensive digital platform for organizing and analyzing agricultural plots. It serves as the central hub for all field related operations, from registration through detailed environmental and agronomic insights.

Field Registration and Location Management. Users initiate field registration via a React Native interface. During creation, the Expo Location API automatically detects and populates GPS coordinates when the "Use Current Location" button is tapped; manual entry remains available for flexibility. Mandatory form fields (field name, area, latitude, longitude) are validated on□device, while optional parameters (soil type, crop type, description) enable further contextualization.

Backend Entity Model and Relationships. On the backend, the `Field` entity—imple-

mented with TypeORM—acts as the primary data hub. It maintains a `ManyToOne` association with `User` and `OneToMany` links to related records: soil analyses, fertilizer applications, crop growth logs, organic pest control entries, and crop rotation plans. Geographic coordinates are stored with decimal precision, area values are recorded in decares, and dynamic weather data are serialized into a JSON column.

Weather Integration. Each field's real□time meteorological data (temperature, humidity, wind speed) are fetched automatically via the `WeatherService`. These parameters are displayed in both the field listing and detail screens to inform agronomic decisions. Robust error handling logic ensures system stability in the event of API failures.

Field List and Detail Views. The frontend employs a `FlatList` component for performant field listings, with each card summarizing location, area, crop type, and current weather. A pull to refresh gesture allows users to manually reload data. The detail view implements tab navigation with "Overview" and "My Sensors" tabs: the former presents field metadata and weather, while the latter enables sensor registration by unique ID and displays manually entered sensor readings alongside historical soil nutrient graphs.

Security and Authorization. All field endpoints are protected by JWT authentication using the `@UseGuards(JwtAuthGuard)` decorator. The `@GetUser()` decorator retrieves the authenticated user context, and ownership validation ensures that each field record is accessible only by its creator.

Error Handling and User Experience. Comprehensive error handling mechanisms including loading indicators, descriptive error messages, retry logic, and `Alert.alert` prompts—optimize usability. Empty state screens guide users when no fields are present, maintaining a clear and user□friendly interface.

This implementation empowers farmers to manage their agricultural plots effectively, combining modern web technologies with a secure, intuitive user experience.

```
1. Field Creation Process
    INITIALIZE form with required fields (name, area, latitude,
       longitude)
    ENABLE optional fields (soilType, cropType, description)
    IMPLEMENT GPS location detection using Expo Location API
    REQUEST foreground location permissions from user
    VALIDATE form inputs with required field checks
```

```
    SUBMIT field data to backend via REST API
    HANDLE success/error responses with user feedback

2. Field Data Management
    STORE field data in TypeORM entity with decimal precision
    ESTABLISH Many-to-One relationship with User entity
    CREATE One-to-Many relationships with related entities
    MANAGE field ownership through user authentication
    HANDLE field updates and deletions with ownership validation
    MAINTAIN data integrity through database constraints

3. Weather Integration
    FETCH weather data for each field using WeatherService
    RETRIEVE temperature, humidity, wind speed, and conditions
    STORE weather data in JSON field format
    HANDLE weather API failures gracefully
    UPDATE weather information on field list refresh
    DISPLAY weather data in field cards and detail views

4. Field Listing and Navigation
    FETCH user-specific fields from database
    DISPLAY fields in FlatList with card-based layout
    SHOW field information (name, location, area, crop type)
    INTEGRATE weather data in field cards
    ENABLE pull-to-refresh functionality
    NAVIGATE to field detail page on card tap

5. Field Detail Management
    LOAD comprehensive field information
    DISPLAY field properties in organized grid layout
    SHOW weather information with current conditions
    PRESENT tab navigation (Overview, Sensors)
    LINK to sensor management for field-specific sensors
    PROVIDE basic field analytics and data
```

```
6. Basic Analytics
    COLLECT field-related data from entities
    ANALYZE basic soil analysis data
    TRACK crop growth measurements
    MONITOR fertilizer application records
    GENERATE simple field analytics
    PRESENT data in structured format

7. Error Handling and User Experience
    IMPLEMENT loading states during data operations
    HANDLE network errors with basic error messages
    VALIDATE user permissions for field operations
    PROVIDE user-friendly error messages
    MANAGE empty states for new users
    ENSURE basic responsive design
```

4.0.9. Frontend Development

This mobile app is developed using React Native and Expo framework to provide a modern and responsive user interface.

4.0.10. UI Design with React Native and Expo

The user interface of the mobile application was designed using React Native and the Expo framework. The following aspects highlight the key design and architectural decisions:

Navigation Structure A tab based navigation model was implemented using Expo Router. It enables fluid transitions between key application views such as the home page, planning, test management, evaluation, crop search, and growth tracking. This modular structure supports scalability and future enhancements.

State Management Local component level state is managed using React's core hooks, including `useState` and `useEffect`. No centralized state management system (e.g., Redux, Zustand, or Context API) was used; instead, each component independently manages its internal state, which is currently sufficient for application requirements.

Reusable Components The interface makes use of several modular UI components, including `Collapsible`, `ExternalLink`, `HapticTab`, `HelloWave`, `ParallaxScrollView`, `ThemedText`, and `ThemedView`. These components enhance visual consistency and facilitate maintainable code development.

4.0.11. User Interface Implementation

The user interface of the TarlaYoldası application was developed using React Native and the Expo framework, offering a modern and intuitive experience aligned with contemporary UI/UX design standards. The declarative structure of React Native facilitated the implementation of interactive and maintainable interface components, with a particular focus on visual clarity, data accessibility, and ease of user interaction.

Typography principles were applied to establish a clear information hierarchy. Readable font families, appropriate text sizing, and consistent heading structures were adopted to promote clarity and accessibility across content layouts. Interaction design was supported through intuitive navigation patterns, haptic feedback, and visual cues, effectively guiding users throughout the application and enhancing usability. Components such as collapsible sections contribute to efficient content organization, allowing users to interact with interface elements based on their needs.

The visual design maintains a clean and modern aesthetic through the use of a consistent color scheme and a layout tailored to agricultural use cases. Reusable components like `ThemedText` and `ThemedView` were utilized to support consistent theme management, enabling dynamic styling across various screens. Additionally, the use of `ParallaxScrollView` enhances the visual structure of the interface and contributes to a smooth and engaging scrolling experience.

From a structural perspective, the application features a tab-based navigation system defined in `(tabs)/layout.tsx`, providing direct access to core modules such as sensor data visualization, growth stage tracking, soil analysis, and fertilizer recommendations. These modules are visually structured to highlight key information and support user decision making processes. Complex data elements, including nutrient values and crop growth patterns, are presented through organized visual layouts that enhance interpretability.

Although performance monitoring tools are in place, advanced optimization strategies such as lazy loading, image optimization, and caching mechanisms have not been implemented. Similarly, while certain interface sections demonstrate a strong visual hierarchy, this design principle

is not applied consistently throughout the application. Accessibility support, including screen reader compatibility, keyboard navigation, and high contrast modes, has not yet been integrated.

In summary, the user interface of the TarlaYoldası application successfully combines usability and visual clarity through the use of modern frameworks and design practices. While there are opportunities for improvement in areas such as accessibility, performance optimization, and consistency in visual hierarchy, the current implementation effectively supports user engagement and agricultural data interaction.

4.0.12. Authentication Screens

Figure 4.1: opening screen

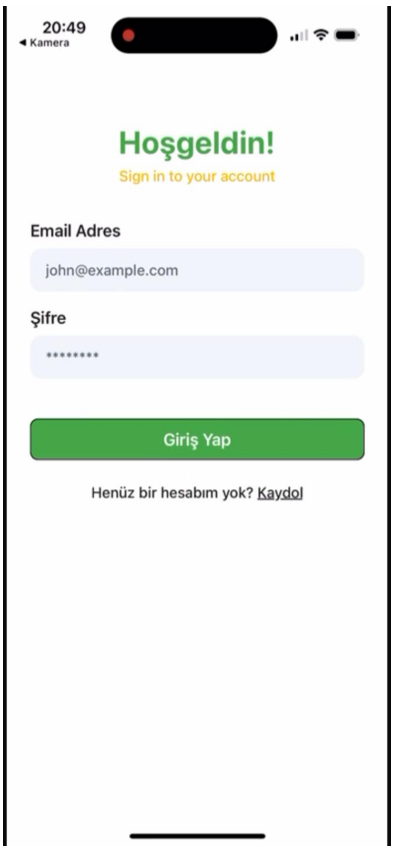


Figure 4.2: log-in screen

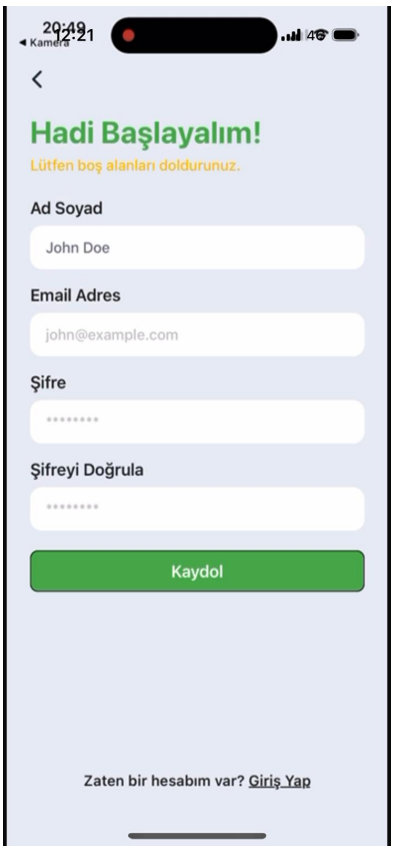


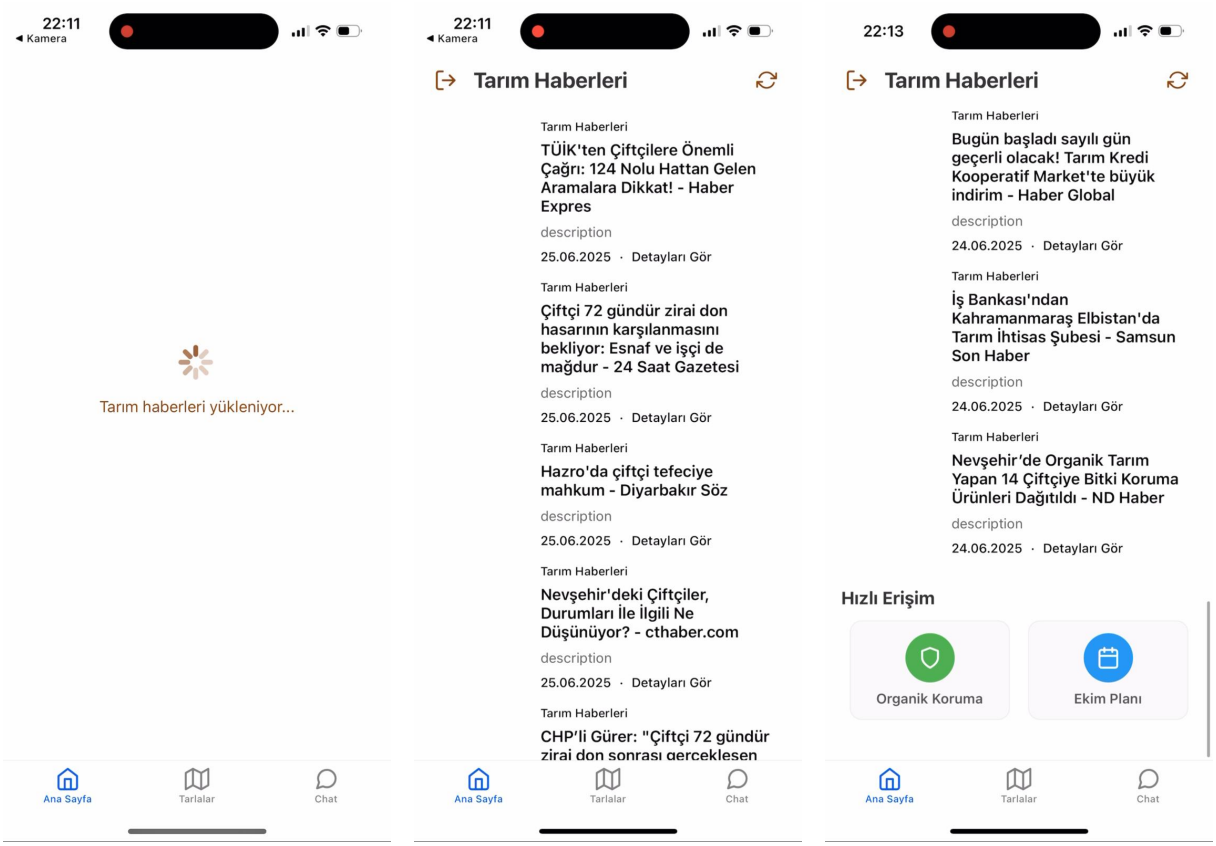
Figure 4.3: sign-up screen

Figure 4.4: Authentication Screens

The authentication system offers a three stage screen flow to ensure secure user access. The splash screen introduces users to the application with a minimalist background and a custom logo that reflects the visual identity of the system. This screen clearly presents the application's name and purpose. The login screen allows users to access the system with their existing accounts. It includes input fields for email and password. Users can enter their credentials and click the "Login" button. For users who do not yet have an account, a "Sign Up" link is provided. This screen communicates with a NestJS based backend API, utilizing JWT (JSON Web Token) for

secure authentication. Passwords are securely hashed using bcrypt, and verification processes are handled on the backend. The registration screen facilitates the onboarding of new users into the system. Basic information (e.g., email and password) is collected from the user and a new user record is created in the database (SQLite) via the backend API. Throughout this process, user data security is prioritized and protected in accordance with modern security standards. On the frontend, React Native and Expo technologies are used to deliver a mobile friendly and user centric interface. Through this tripartite structure, the application ensures both a seamless user experience and compliance with contemporary security requirements.

4.0.13. News Screen



(a) News Loading Screen (b) NEWS SCREEN (c) Access to another page

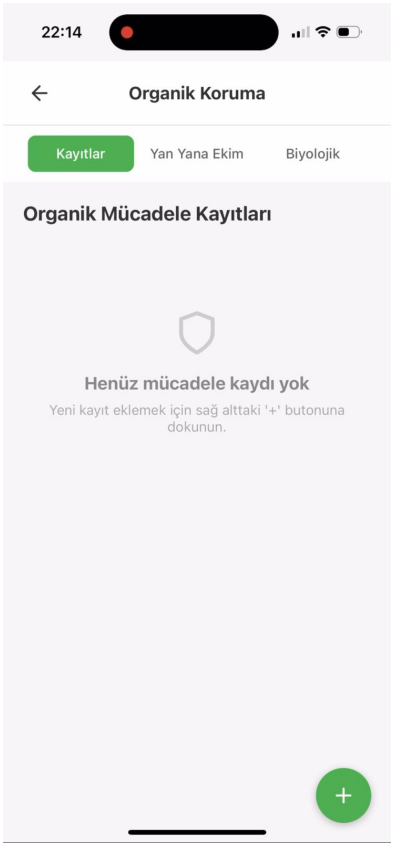
The agricultural news module of the mobile application was designed to provide users with seamless access to up-to-date sectoral developments while ensuring a high level of responsiveness and user engagement. Upon launching the application or during data retrieval processes, a centered loading indicator (ActivityIndicator) accompanied by an informative message (”Loading agricultural news...”) is displayed, effectively preventing user confusion caused by empty or unresponsive screens. This visual feedback is maintained within a clean and minimal interface, with persistent access to core navigation via a bottom tab bar. Once the data is fetched from the backend API (e.g., via the `/news` endpoint), the loading state transitions and the retrieved news

articles are rendered using optimized components such as `FlatList` or `ScrollView`, supporting smooth vertical scrolling. Each article is presented within a structured card format, displaying key elements such as category labels, prominent headlines, concise descriptions, publication dates, and a "View Details" link that enables navigation to in-depth views or external web content via WebView. To enhance usability and cross-module engagement, quick-access cards are integrated, directing users to relevant modules such as Organic Protection or Planting Plan through interactive `TouchableOpacity` components. The news screen also incorporates a refresh icon allowing users to manually trigger data re-fetching. Error handling mechanisms are embedded to inform users of connectivity or API-related issues, along with a retry option. The overall layout adheres to modern UI/UX principles, employing a consistent color palette, responsive typography, and scalable design that ensures accessibility across diverse screen sizes. All components are implemented using contemporary React Native and Expo frameworks, ensuring a balance of performance, maintainability, and a user-centered experience.

4.0.14. Organic Preservation Screen

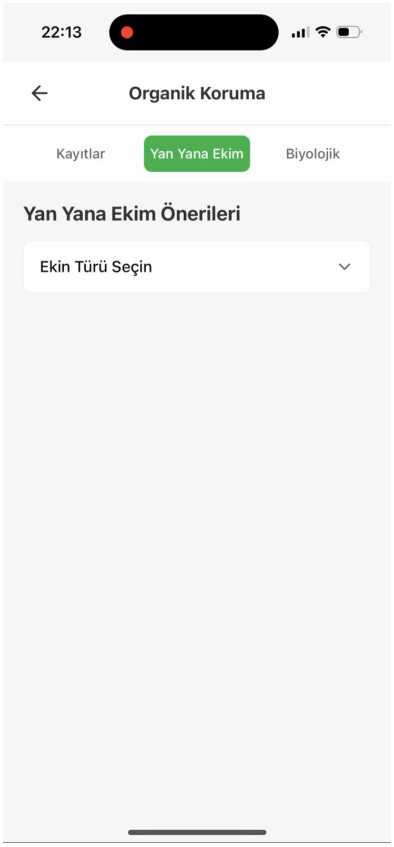
The Organic Protection module of the mobile application has been meticulously designed to enhance user experience by providing intuitive access to pest control records and recommendations. During data retrieval processes, a centered loading animation (ActivityIndicator) accompanied by an informative message such as "Loading organic protection data..." is displayed within a clean and distraction-free interface. This prevents confusion and ensures the user perceives the system as responsive. Persistent bottom tab navigation offers continuous access to core sections like Home, Fields, and Chat. Upon successful API communication with endpoints such as `/organic-pest-control`, `/organic-pest-control/companion-planting`, and `/organic-pest-control/biological-control`, the retrieved data are rendered within well-structured lists using `ScrollView` or `FlatList` components. Users can manually refresh data via a refresh icon located in the top-right corner, which re-triggers the API request and associated loading state. The interface includes a tab-based navigation structure that categorizes content into sections like "Records," "Companion Planting," and "Biological," each fed by different data sources. Each entry in the list presents detailed agronomic information such as pest names, control methods, efficacy rates, descriptions, and dates, along with category tags (e.g., Biological, Cultural, Physical) for enhanced clarity. Users may tap on any item to access more detailed information via either a modal view or a dedicated screen. To facilitate cross-module interaction, quick-access cards are included, allowing seamless transitions to other relevant parts of the application. Furthermore, users are able to add new pest control entries through a form accessible via a floating "+" button. The visual design emphasizes trust and clarity by employing natural color schemes (e.g., green, blue, and orange tones), bold typographic hierarchies for key information, and fully responsive layouts that adapt to a variety of screen sizes. Through

[h]

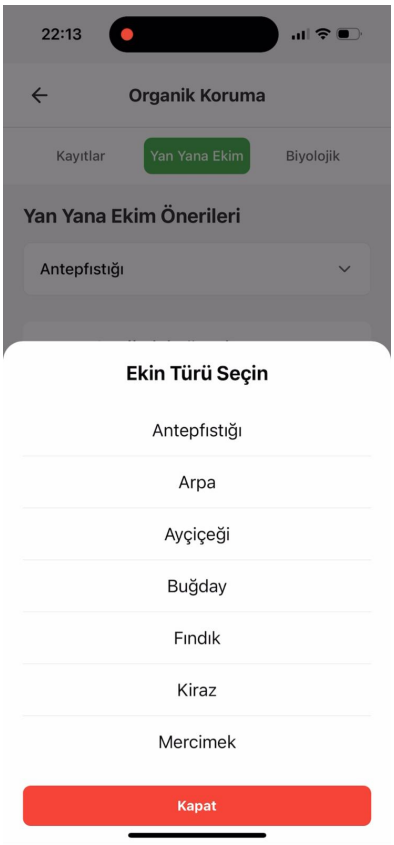


(a) Organic preservation screen

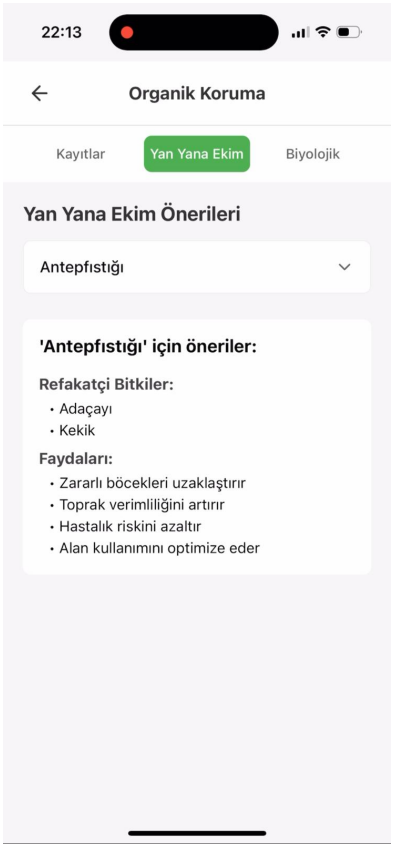
(b) Register Detail Screen



(c) side by side planting recommendation



(d) choose crop screen for organic preservation



(e) Suggestion Screen

Figure 4.6: Organic Preservation

the integration of these features, the module ensures an effective, user-centered interface for managing organic pest control strategies.

4.0.15. Chat screen

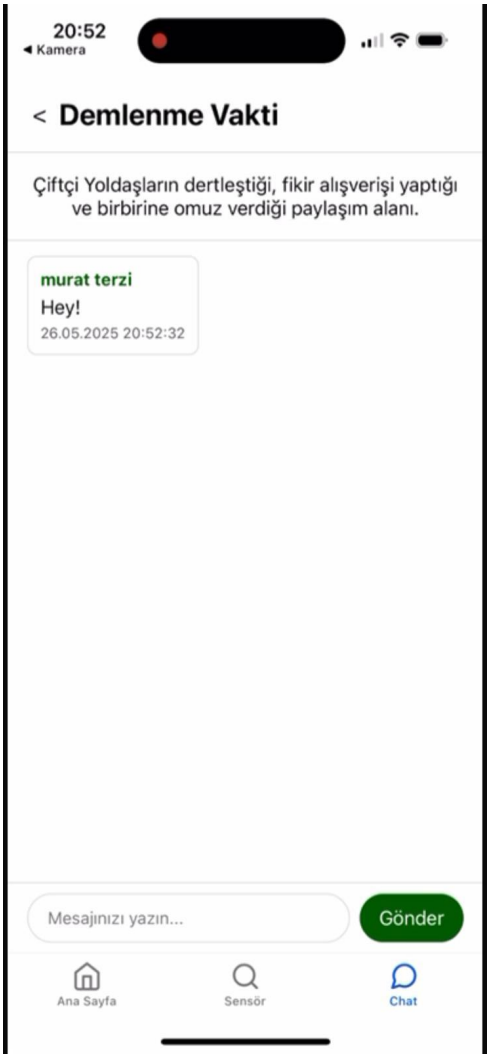
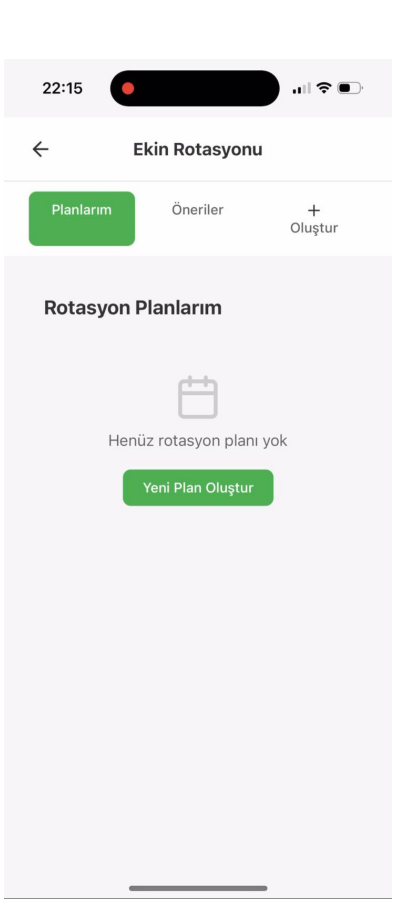


Figure 4.7: Chat Screen

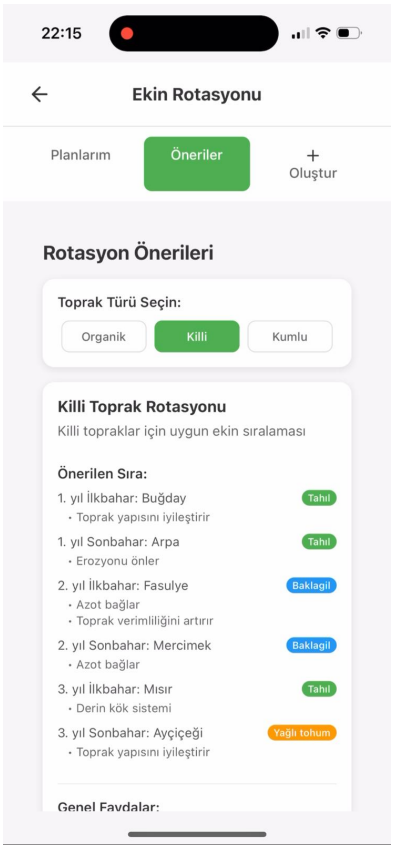
The application provides a dedicated communication platform designed primarily for farmers, enabling them to exchange ideas, seek advice, and engage in written discussions. This interactive sharing space, titled “Demlenme Vakti”, is introduced with a header and a brief explanatory text that outlines its purpose, positioned at the top of the screen. The core feature of this interface is the message list, where user-generated messages are displayed as individual cards containing the sender’s name (e.g., “Murat Terzi”), message content, and the timestamp indicating when the message was sent. These messages are most likely retrieved from the backend via API calls and rendered in chronological order. At the bottom of the screen, a text input box accompanied by a “Send” button allows users to compose and submit new messages. Upon pressing

the “Send” button, the message is sent to the backend through a POST request, and upon successful submission, it is immediately appended to the message list on the frontend. The input field is then cleared to facilitate further messaging. Navigation is facilitated by a bottom tab bar, which allows seamless switching among core application sections, namely Home, Sensor, and Chat.From a technical perspective, the frontend leverages React Native, Expo, and React Navigation frameworks to deliver a responsive and mobile-friendly user interface. Message data is managed via state variables; messages are fetched from the API and stored in state, enabling real-time updates when new messages are sent. The system supports near-instantaneous message exchange, enhancing user interaction. Each message card clearly displays the sender’s identity and timestamp to maintain conversational context. Overall, the interface is designed with simplicity and clarity in mind to provide an intuitive and user-centric communication experience.

4.0.16. Crop Rotation



(a) Crop rotation Plan Screen



(b) Recommendations according to soil type Screen

(c) Register Detail Screen For Crop rotation

The Crop Rotation module of the mobile application has been meticulously designed to offer users both flexibility and scientific guidance in managing their agricultural planning. The interface features a tab-based navigation structure consisting of three main sections: *My Plans*,

Recommendations, and *Create*. In the *My Plans* tab, users can view previously created rotation plans or initiate the creation of a new one. If no existing plans are found, an informative icon and message (e.g., "No rotation plans yet") are displayed alongside a call-to-action button to encourage plan creation.

The *Create* tab provides a step-by-step form allowing users to define a multi-year rotation plan tailored to their specific field conditions. Required inputs include the plan name and rotation duration, while each year can be populated with crop name, crop family, and agronomic properties such as nitrogen fixation, pest repellent effects, and soil improvement capabilities, selectable via switch buttons. Additional years can be dynamically added using an "Add Year" button, and plans are submitted to the backend via API upon clicking "Create Plan," after which users are redirected to the *My Plans* tab.

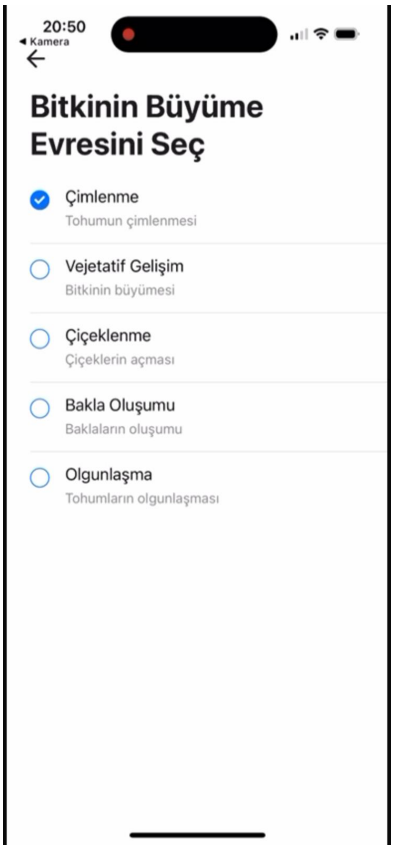
The *Recommendations* tab presents scientifically backed crop rotation sequences based on the user's selected soil type (e.g., organic, clay, sandy). These recommendations include concise agronomic justifications and are visually enriched with color-coded crop category tags (e.g., *Cereals*, *Legumes*, *Oilseeds*). All forms are managed through stateful structures that support dynamic field addition and real-time updates. The system handles both frontend-rendered and API-driven data depending on the configuration, enabling adaptability and offline support.

Feedback mechanisms are integrated throughout, including placeholder messages in empty states and success confirmations upon plan creation. The visual design adopts a modern and accessible style, leveraging natural colors such as green, blue, and orange to highlight categories and active elements, while ensuring responsiveness across various screen sizes. This comprehensive user experience is powered by React Native and the Expo framework, emphasizing performance, scalability, and ease of use in promoting sustainable agricultural planning.

4.0.17. Fertilizer-recommendation screen

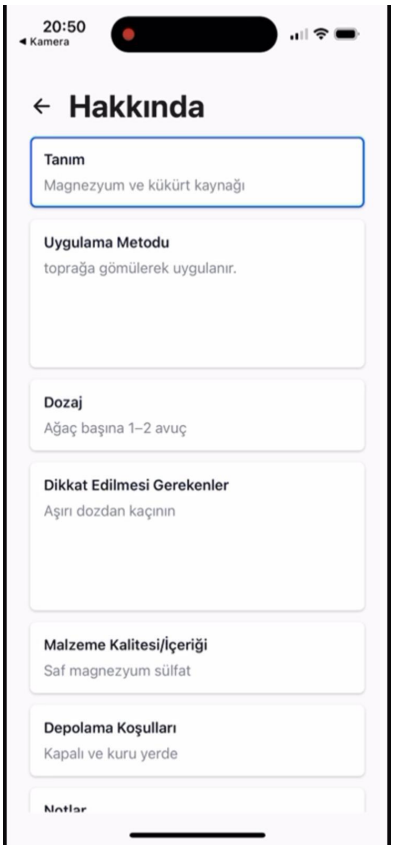
Crop Type Search and Selection Screen Following the soil analysis process, the user is guided to select the appropriate crop type. This screen displays a list of crop types, which can be filtered using a search input. The implementation is handled in the frontend/app/ekinsearch.tsx file. Crop type data is dynamically retrieved from the backend API via the /crop-types endpoint. Each crop type is displayed as a card that includes its name and a brief description. As the user types in the search box, the crop list is filtered in real-time to match the input query. When a crop type is selected, the application navigates to the growth stage selection screen, passing the selected crop type's identifier as a parameter. Plant Growth Stage Selection Screen In the next step, users select the current growth stage of the chosen crop. Growth stages include key phases

(a) Crop Type



(b) Choose growth phase

(c) Choose Plan



(d) Fertilizer Detail Screen

Figure 4.9: fertilizer recommendation system

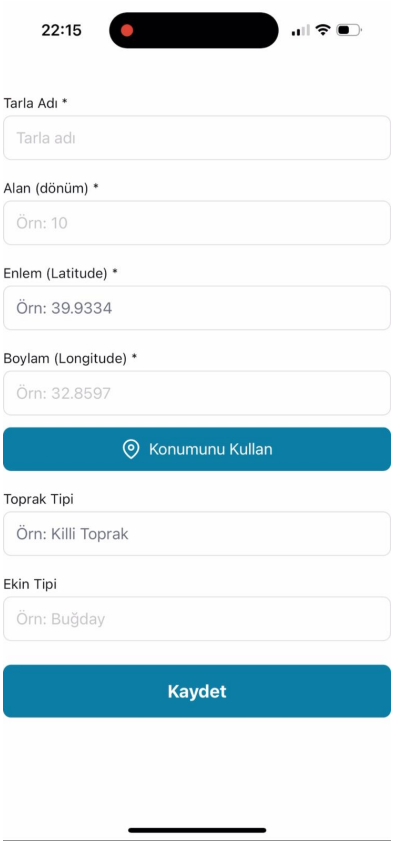
such as germination, vegetative growth, flowering, pod formation, and maturation. This feature is implemented in the frontend/app/buyume.tsx file. Based on the previously selected crop type, the system fetches the relevant growth stages from the API endpoint /growth-stages/by-crop-type/:cropTypeId. Each stage is presented as a radio button, accompanied by its name and a short explanation. Upon selecting a growth stage, the user is redirected to the organic fertilizer or soil amendment recommendation screen, where tailored suggestions are presented. Plan Selection (Organic Fertilizer Recommendation) Screen The final stage allows the user to choose an appropriate organic fertilizer or soil amendment plan based on the selected crop type, its growth stage, and the user's soil analysis results. This screen is implemented in the frontend/app/plan.tsx file. The system sends a POST request to the /fertilizer-rules/recommendations endpoint, including parameters such as crop type, growth stage, and relevant soil data. In response, the API returns a list of recommended fertilizer plans. Each plan is displayed as a card that contains the name and a brief description of the recommendation. When a user selects a specific plan, they are redirected either to a detailed view or to the subsequent step in the agricultural workflow.

4.0.18. Field Screen

The Field and Sensor modules of the TarlaYoldası application were designed to ensure both intuitive user interaction and seamless support for real-time sensor integration. In the Field module, users can view a list of registered agricultural fields, each presented with its name, area, location, and a brief description. Selecting a field navigates the user to a detail view that includes metadata such as soil type, current crop, sensor information, weather data, and historical analysis results. This interface utilizes a tab-based structure (e.g., "Overview", "My Sensors") to streamline navigation. If no fields exist, a placeholder message and a call-to-action button are displayed, encouraging users to add their first field. Data is fetched via the `/fields` endpoint and managed locally in component state, while creation and updates are executed through POST/PUT requests.

The Sensor module complements this functionality by allowing users to simulate sensor behavior during development, while preserving full compatibility with physical IoT devices for future integration. This interface-first, mock-driven approach supports extensibility by enabling HTTP POST submissions to the `/sensors` endpoint with unique sensor IDs. Sensors are linked to authenticated users using JWT-secured endpoints, ensuring only authorized devices or clients can submit or retrieve data. Each sensor card displays the latest measurement, sensor type, and update timestamp, while detailed views provide comprehensive metrics such as temperature, humidity, pH, nitrogen, phosphorus, and potassium levels. Real-time updates are supported through additional endpoints like `/sensors/:sensorId/update`, which also main-

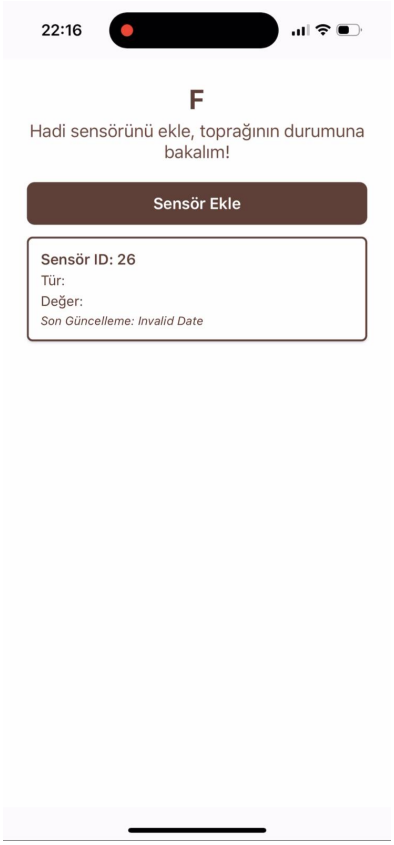
(a) Add Field Screen



(b) Enter Location Detail Screen

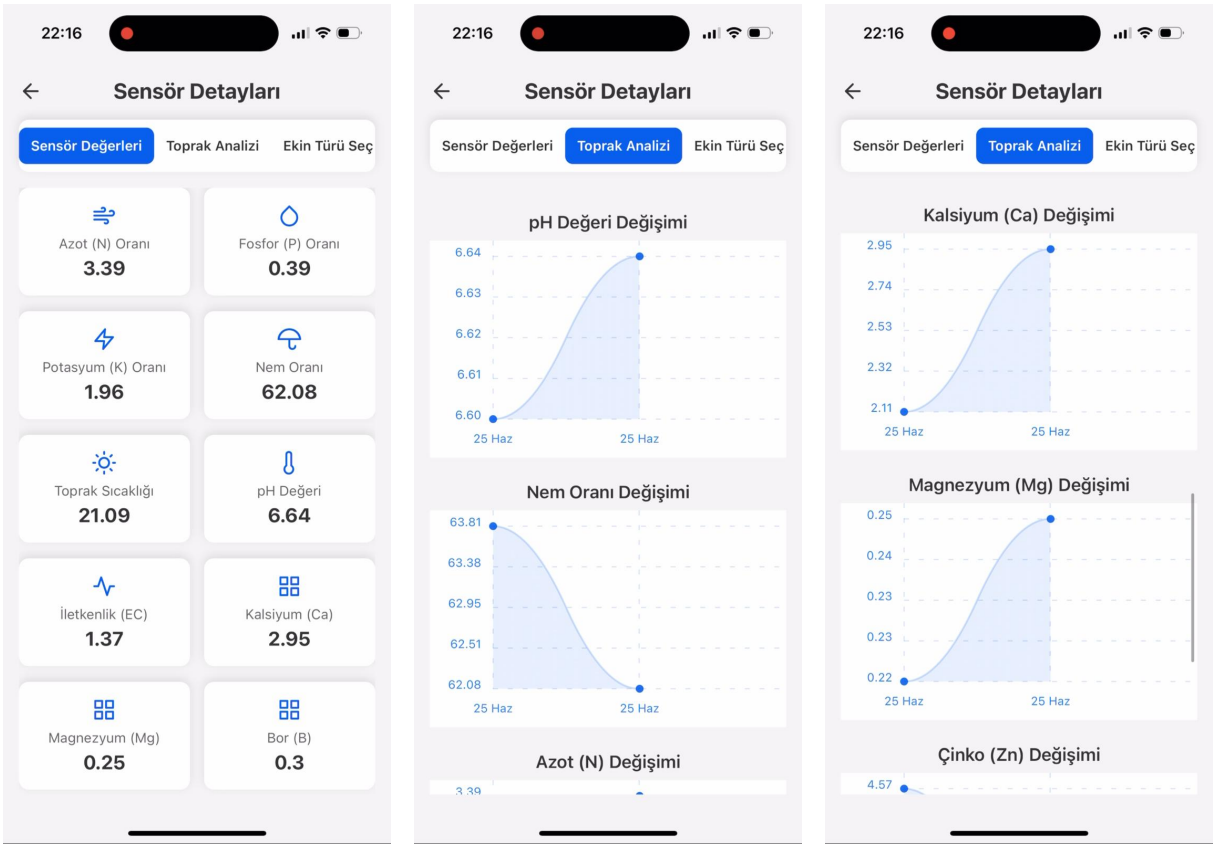
(c) Field Detail Screen

(d) Add sensor screen



(e) Sensor Id

Figure 4.10: Field and Sensor Registration Interface



(a) Soil Values (b) Soil Analysis (c) Soil Analysis Screen

Figure 4.11: Detailed Soil Parameter Visualizations

tain a historical log in a separate database table. Sensory data is visualized using interactive cards, enhancing interpretability and accessibility.

Sensors can be associated with specific fields (`field_id`) and fetched through endpoints such as `/sensors/field/:fieldId`. To maintain security and data integrity, all sensor-related operations are protected using `@UseGuards(JwtAuthGuard)`, and JWT tokens are required in the Authorization header for each request. The backend verifies these tokens and restricts access based on user credentials. Additional endpoints, such as `/sensors/connect/:sensorId` and `/sensors/disconnect/:sensorId`, allow users to manage ownership and connectivity. Only one user can be associated with a sensor at any time, preventing conflicts, and sensors can be reassigned after disconnection. Furthermore, data validation is enforced via DTOs and class validators, and each sensor entity is designed to be extensible, allowing the addition of new nutrient parameters without disrupting the existing architecture. In the field registration process, users can obtain their geographic coordinates automatically by tapping the "Use My Location" button. Upon activation, the application prompts the device for location permissions and retrieves the current GPS coordinates using the `expo-location` library. The acquired latitude and longitude values are then programmatically inserted into the form fields and submitted to the backend as part of the field creation payload. This mechanism eliminates the need for manual entry and enhances the accuracy of spatial data.

Furthermore, weather information is dynamically integrated into the field detail screen. Based on the saved latitude and longitude, the system sends a request to an external weather API (e.g., OpenWeatherMap) either from the backend or frontend. The API response, containing real-time meteorological parameters such as temperature, humidity, and wind speed, is then rendered in the user interface. This integration provides users with actionable insights by aligning agronomic decisions with localized climatic conditions. The implementation effectively combines device-level geolocation with external data sources, contributing to the overall precision and usability of the agricultural decision-support system.

4.0.19. User Flow and Interaction Workflow

The frontend application follows a structured workflow to handle user interactions efficiently:

```
START

User launches application
```

```
IF network_is_connected():
    sync_data_with_cloud()
ELSE:
    load_cached_data()

show_main_menu()

User selects feature:
    CASE "Ekin Arama":
        open_crop_search_module()
    CASE "Büyüme Takibi":
        open_growth_tracking_module()
    CASE "Değer Analizi":
        open_value_analysis_module()
    CASE "Plan Oluşturma":
        open_plan_creation_module()
    CASE "Test Yönetimi":
        open_test_management_module()

process_user_request()
update_UI()

show_recommendations_and_insights()

END
```

4.0.20. Navigation and State Management Architecture

Navigation

The mobile application adopts a modern file-based routing architecture facilitated by the `expo-router` library. All user interface screens are rendered within a unified layout structure defined in `frontend/app/layout.tsx`, following the *Single Activity / Single Layout* paradigm. This architectural choice ensures coherent lifecycle management and seamless state transitions.

At the initial entry point (`frontend/app/index.tsx`), a splash screen is presented to the user. Upon completion, the system redirects to the `/auth/signin` route using programmatic navigation. Following successful user authentication, dynamic route transitions are managed

by the routing system, which supports context-aware flow control and enhances the overall user experience.

colorstack□pop4.0.20.1. State Management. Persistent user session handling is achieved through the `@react-native-async-storage/async-storage` library, which securely stores authentication tokens on the client device. These tokens are automatically appended to the `Authorization` header of each HTTP request, ensuring secure and authenticated communication with backend services.

A centralized API service layer is implemented in `frontend/app/services/api.ts`, which abstracts network operations (e.g., `GET`, `POST`, `PUT`, `DELETE`). This service layer handles error propagation, response standardization, and request formatting, thus promoting consistency and reusability across the application.

State synchronization between screens is ensured via consistent API calls and interaction with local storage, allowing real-time data access and minimizing discrepancies regardless of the navigation path.

Technical Architecture Highlights

- **Single Activity / Single Layout Paradigm:** All components are rendered within a unified layout, simplifying lifecycle monitoring and state handling.
- **Dynamic Routing:** Real-time route decisions are made based on user interactions, taking advantage of the reactivity provided by `expo-router`.
- **Offline Support:** Essential data, such as session tokens and frequently accessed resources, are cached locally to enable offline functionality.
- **Centralized API Layer:** A shared API service streamlines authentication, data transactions, and error management in a centralized manner.
- **Component-Based UI Architecture:** The interface is developed using modular React Native components (e.g., `Collapsible`, `ParallaxScrollView`, `ThemedText`), enhancing code reusability and maintainability.

4.0.21. Backend Development

The backend infrastructure of the Yepyeni project is built using the NestJS framework with TypeScript, integrating a modular architecture and SQLite database via TypeORM. Each functional domain (e.g., users, sensors, crops, fertilizers) is encapsulated in separate modules and entities, ensuring clear separation of concerns. The API follows RESTful principles through controllers, while business logic is handled in dedicated services. Authentication is implemented using JWT, and database schema management is supported via migrations. The architecture enforces type safety, validation, and error handling, and remains portable across SQL-based databases—making it a scalable, secure, and maintainable solution for agricultural data systems.

4.0.22. Auth Module: Authentication & Authorization

The authentication infrastructure of the system is built using modern security practices integrated into the Auth module of the NestJS framework. User credentials are securely managed through hashed passwords using bcrypt, while session validation and access control are handled via JSON Web Tokens (JWT). Authentication workflows are encapsulated in auth.service.ts and exposed through auth.controller.ts, enabling login, registration, and token issuance functionalities. The system employs guards such as jwt-auth.guard.ts and local-auth.guard.ts to protect API endpoints and enforce role-based access control. Upon successful login, a JWT token is generated and attached to subsequent requests via the Authorization header, where it is validated before granting access. This modular structure ensures secure user management, consistent identity verification, and reliable authorization across the backend architecture.

```
// Authenticate User
function authenticateUser(email, password):
    user = findUserByEmail(email)
    if user is null or !verifyPassword(password, user.hashedPassword):
        return error("Invalid credentials")
    token = generateJWT(user.id)
    return success(token)

// Authorize Request (JWT Guard)
function authorizeRequest(token):
    if !isValidJWT(token):
        return error("Unauthorized")
    userId = decodeJWT(token)
    return success(userId)
```

4.0.23. Sensors Module: Sensor Data Management

```
// Add Sensor Data
function addSensorData(sensorId, data):
    if !isValidSensor(sensorId):
        return error("Sensor not found")
    saveSensorDataToDatabase(sensorId, data)
    return success("Data saved")

// Get Sensor Data
function getSensorData(sensorId, dateRange):
    data = querySensorData(sensorId, dateRange)
    return success(data)
```

The sensor data handling mechanism in the system encompasses secure ingestion, querying, and access control processes. Sensor devices or authenticated users transmit environmental measurements (e.g., temperature, humidity, soil moisture) via API, specifying a registered sensor ID. The backend verifies the validity of the sensor and input data before persisting the information in the corresponding database table. Historical sensor data can be queried based on sensor ID and date range, with the backend retrieving the relevant entries and returning them in JSON format. All operations are protected by JWT-based authentication, ensuring that only authorized users can access or modify their own sensor data, thereby maintaining data integrity and secure access control across the platform. Operations related to sensor management—such as adding, linking, unlinking, and listing sensors—are executed strictly based on user authentication. Each API request is secured using a JWT (JSON Web Token) mechanism, and upon successful verification, the authenticated user's identifier is automatically attached to the request object via req.user.id. When a new sensor is added, it is directly associated with the authenticated user. For linking or unlinking operations, the system verifies that the targeted sensor belongs to the requesting user; if not, the request is rejected with an appropriate error response. Similarly, sensor listing functions are restricted to return only the sensors owned by the currently authenticated user. This access control model ensures that users can only interact with their own data, thereby preserving data privacy and maintaining a high standard of security across the system.

4.0.24. Forum Messages Module: Forum Messaging

```
// Post Forum Message
function postForumMessage(userId, messageContent):
    if !isValidUser(userId):
```

```
        return error("User not found")
    message = saveForumMessage(userId, messageContent)
    return success(message)

// Get Forum Messages
function getForumMessages(threadId):
    messages = queryMessagesByThread(threadId)
    return success(messages)
```

In the forum messaging system, each message is persistently linked to the authenticated user, ensuring accountability and secure access control. This linkage is established during message creation by extracting the user ID from the validated JWT token and associating it with the message in the database. Users can post, retrieve, edit, or delete messages within topic-specific threads, but modification and deletion operations are permitted only after ownership verification, maintaining data integrity. To enforce input correctness and prevent malformed data submissions, the system utilizes Data Transfer Objects (DTOs) alongside the class-validator library. These tools ensure that critical fields such as message title and content conform to predefined constraints, thereby increasing the robustness of the backend and minimizing runtime validation errors. This structured and secure approach guarantees that only authorized users can interact with their own content while maintaining consistency and integrity throughout the messaging workflow.

5. Testing and Evaluation

5.0.1. Performance,Soak,Memory Test

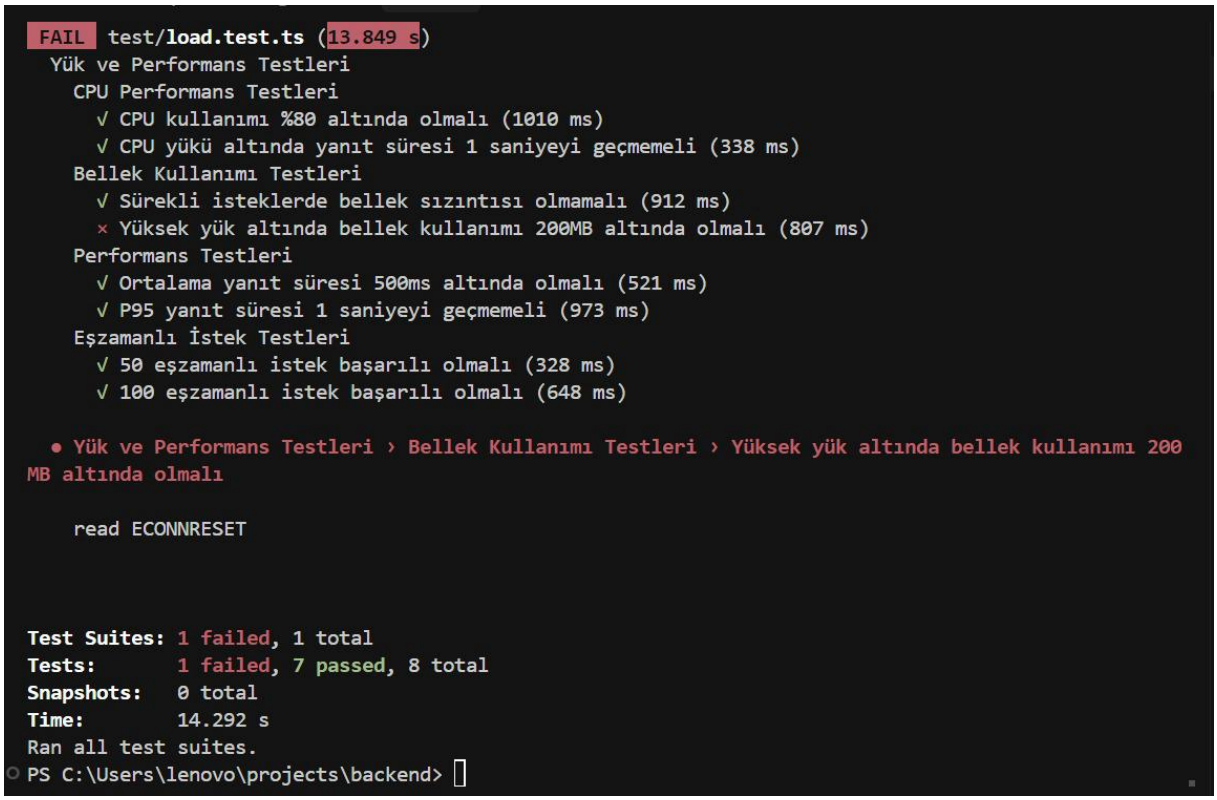


Figure 5.1: Performance Test

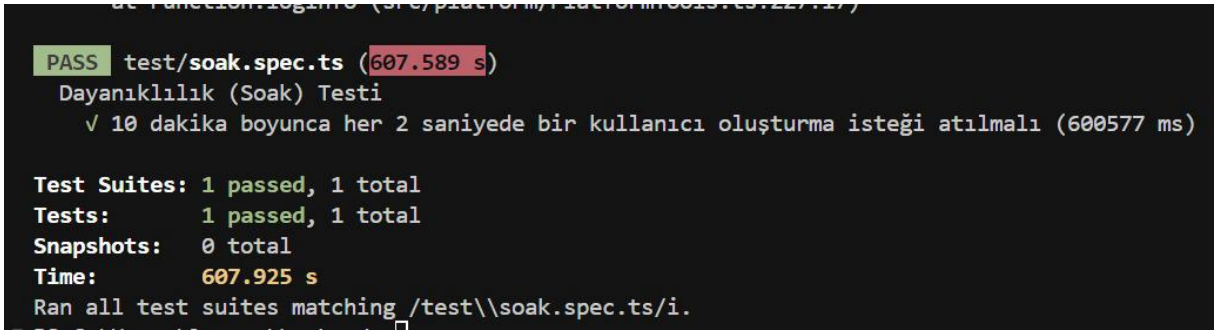


Figure 5.2: Soak test

The results of the load and performance evaluations indicate that the system largely satisfies the expected operational benchmarks under both standard and concurrent usage conditions. CPU performance assessments confirm that processor utilization remains below 80%, with response times consistently maintained under one second even during peak loads. Memory diagnostics reveal no evidence of leakage during sustained request cycles; however, memory consump-

tion under high-load conditions exceeds the 200 MB threshold, suggesting the need for further optimization in resource handling. Performance testing further demonstrates that the system maintains an average response time below 500 milliseconds and a 95th percentile (P95) latency below one second—both of which align with acceptable performance thresholds. Additionally, the system successfully processes 50 and 100 concurrent requests without failure, confirming its robustness in handling simultaneous operations. Despite the overall stability and responsiveness, elevated memory usage under stress conditions highlights a potential area for architectural or implementation-level refinement to enhance scalability and resource efficiency.

The results of the soak test demonstrate the system's ability to maintain stable performance under sustained load conditions. During the test, user creation requests were sent every two seconds over a continuous period of ten minutes. The system successfully handled all requests without failure or interruption, indicating robust stability and resource management over extended operation. This outcome suggests that the application is capable of supporting prolonged usage scenarios without experiencing performance degradation, memory leaks, or service outages. Such resilience under long-term load is a critical indicator of the system's reliability and suitability for real-world deployment.

The results of the stress test reveal a critical limitation in the system's ability to handle extreme concurrent load. During the test, 1,000 simultaneous user creation requests were issued to the application; however, the system failed to process these requests successfully, resulting in connection errors (ECONNREFUSED). This outcome indicates that the current system configuration or infrastructure is not sufficient to support very high levels of concurrency, leading to service unavailability under peak load conditions. Such findings highlight the need for further optimization in terms of scalability, resource allocation, and possibly load balancing to ensure reliable operation during periods of intense demand.

The results of the load and performance tests indicate that the system's API endpoints operate efficiently and reliably under both normal and moderately high load conditions. All basic API functionality tests passed, with response times well below 100 ms for standard requests. Performance and load tests further demonstrate that the system consistently delivers responses within 500 ms, even when subjected to simultaneous requests. Under stress conditions, the application successfully handled 100 concurrent requests with an average response time below one second, and maintained memory usage under 100 MB during continuous operation. These outcomes collectively suggest that the system is well-optimized for typical usage scenarios, exhibiting both low latency and stable resource consumption. Such results confirm the system's readiness for deployment in environments where moderate concurrency and sustained operation are required.

- **Horizontal Scalability**
  - Introduce auto-scaling policies (e.g. Kubernetes HPA) to dynamically add application instances under high load.
  - Employ load balancers to distribute traffic evenly across replicas.
- **Database Optimization**
  - Analyze slow queries via profiling tools and add appropriate indexes.
  - Implement connection pooling and tune pool sizes to prevent resource contention.
- **Caching and CDN Integration**
  - Cache frequently accessed data (e.g. news feed, static lookups) using Redis or in-memory caches.
  - Offload static assets (images, scripts) to a CDN to reduce server load and latency.
- **Asynchronous Processing**
  - Delegate heavy computations (e.g. sensor data aggregation) to background workers with a message broker (RabbitMQ, Kafka).
  - Use job queues to smooth out bursty workloads and guarantee sub-second API responses.
- **Resource Profiling & Hot-Path Tuning**
  - Integrate observability tools (e.g. Grafana, Prometheus) for CPU/memory profiling of critical endpoints.
  - Optimize identified hot paths in code (e.g. replace synchronous loops with streaming or vectorized operations).
- **Circuit Breakers & Rate Limiting**
  - Implement circuit breaker patterns to prevent cascading failures under overload.
  - Apply fine-grained rate limits per endpoint to throttle abusive clients and guarantee fair resource usage.
- **Continuous Performance Testing**

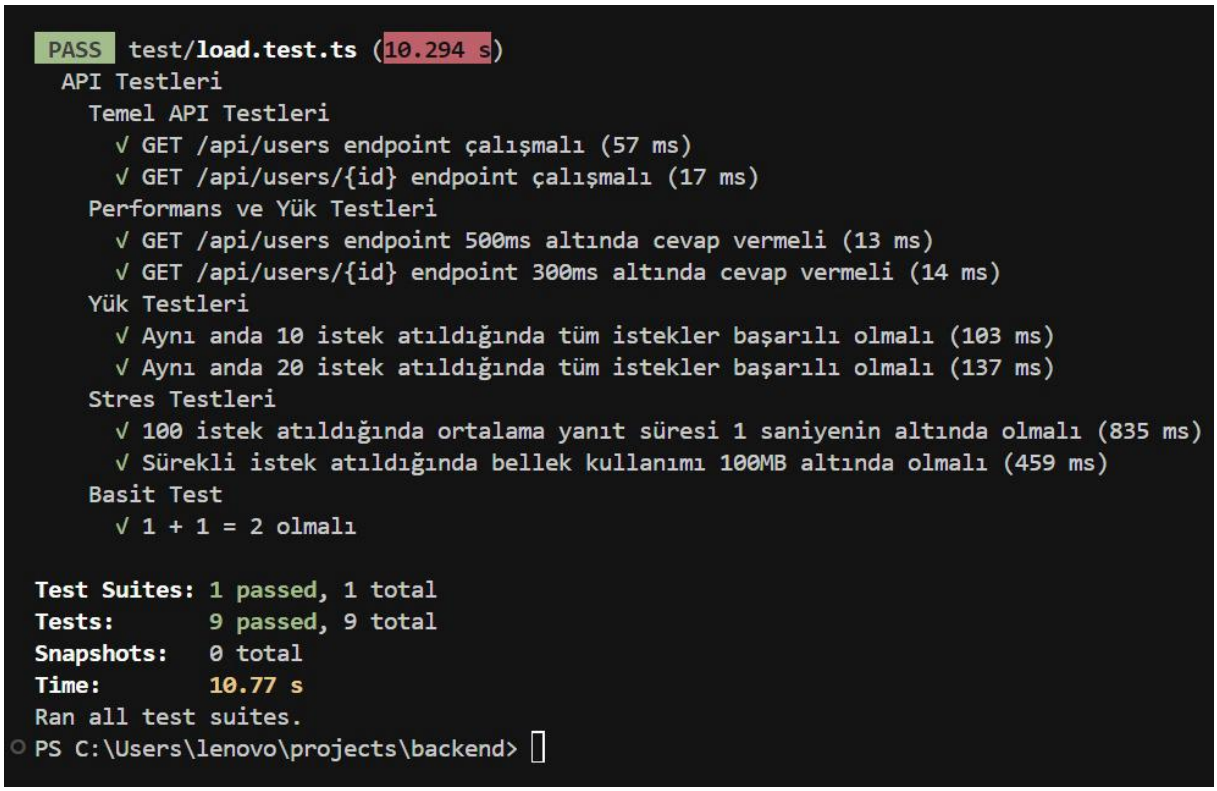


Figure 5.4: Test Mix

– Embed performance benchmarks in CI/CD pipeline to catch regressions early.

– Automate monthly soak tests and stress tests to validate sustained and peak-load stability.

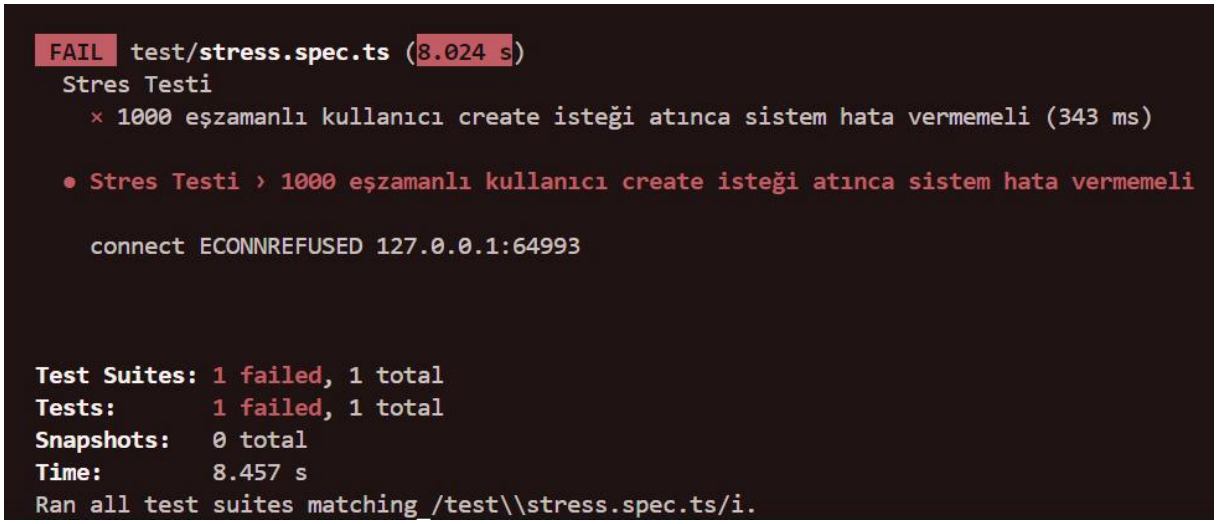


Figure 5.3: Stess Test

5.0.2. Security Test

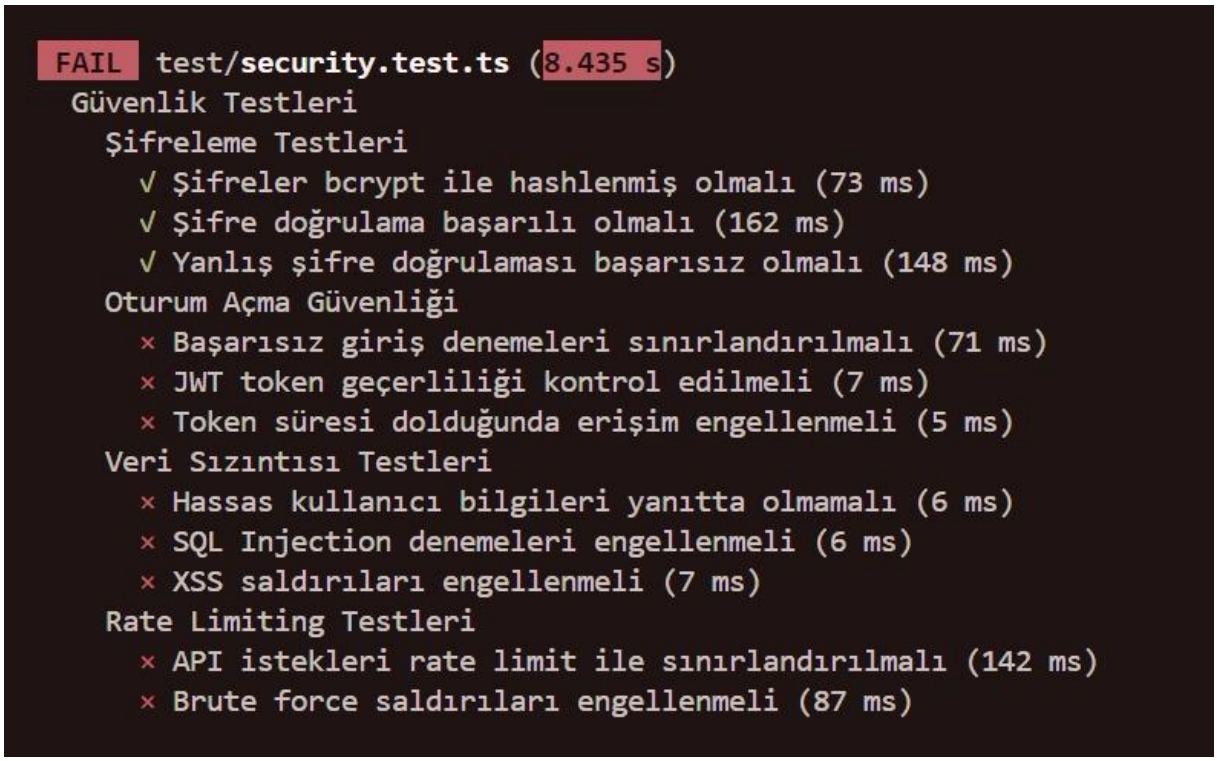


Figure 5.5: Security Test

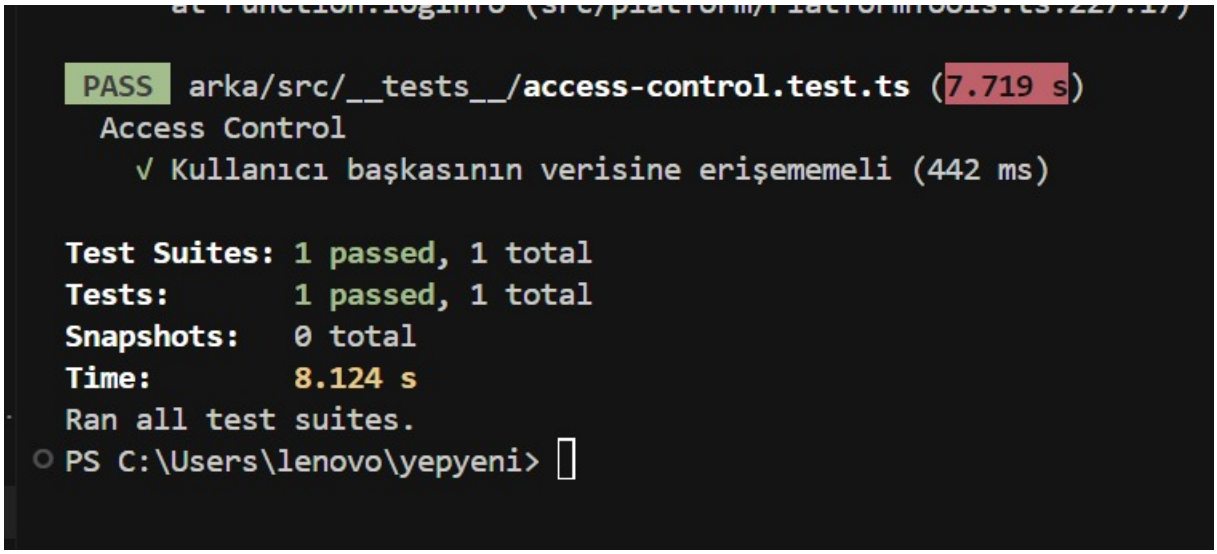


Figure 5.6: Access Control Test Result

The results of the access control and security tests provide valuable insights into the system's current security posture. Access control tests confirm that users are effectively prevented from accessing data belonging to other users, demonstrating the correct implementation of authorization mechanisms. Security tests reveal that password handling is robust, with all passwords

securely hashed using bcrypt and proper validation of both correct and incorrect credentials. However, several critical areas require improvement. The system currently lacks protections against repeated failed login attempts, does not enforce JWT token validity or expiration, and is vulnerable to potential data leakage in responses. Furthermore, essential safeguards against SQL injection, cross-site scripting (XSS), and brute force attacks are not fully implemented, and API rate limiting is absent. These findings indicate that, while foundational security practices such as password hashing and basic access control are in place, the system must be further enhanced to address advanced security threats and ensure comprehensive protection against common attack vectors.

Future Improvements

- **Brute-Force Protection & Rate Limiting**
  - Introduce account lockout or exponential back-off mechanisms following successive failed login attempts.
  - Enforce global and per-endpoint rate limits using token-bucket or leaky-bucket algorithms to throttle excessive requests.
- **Robust JWT Management**
  - Require signature validation and expiration checks for every JWT on incoming requests.
  - Employ short-lived access tokens alongside refresh tokens to balance security and user experience.
- **Comprehensive Input/Output Sanitization**
  - Utilize ORM parameterized queries or explicit escaping to eliminate SQL injection threats.
  - Sanitize all user inputs and HTTP responses to mitigate XSS and prevent data leakage.
- **Automated Security Testing Integration**
  - Incorporate tools such as OWASP ZAP into the CI/CD pipeline for continuous vulnerability scanning and penetration testing.

- Develop targeted test suites for injection attacks, header manipulation, and session hijacking scenarios.

- **Audit Logging & Real-Time Monitoring**
  - Establish centralized logging for authentication/authorization failures and security exceptions.
  - Monitor logs in real time for anomalous patterns and configure alerting for potential breaches.

5.0.3. Unit Test

```
> backend@0.0.1 test
> jest growth-stages.service

 PASS  src/growth-stages/growth-stages.service.spec.ts (5.269 s)
  GrowthStagesService
    √ should be defined (11 ms)
    create
      √ yeni growthStage eklemeli (2 ms)
    findAll
      √ tüm growthStage kayıtlarını döner (2 ms)
    findOne
      √ ID ile growthStage döner (2 ms)
    update
      √ ID ile growthStage günceller (2 ms)
    remove
      √ ID ile growthStage siler (1 ms)

Test Suites: 1 passed, 1 total
Tests:       6 passed, 6 total
Snapshots:   0 total
Time:        5.609 s
Ran all test suites matching /growth-stages.service/i.
```

Figure 5.7: Unit Test

```
PASS src/organic-fertilizers/organic-fertilizers.service.spec.ts (5.82 s)
  OrganicFertilizersService
    findAll
      √ tüm gübreleri döner (13 ms)
    findOne
      √ ID ile gübre döner (3 ms)
      √ Gübre bulunamazsa NotFoundException fırlatır (12 ms)
    getRecommendedFertilizers
      √ kurala uyan gübreleri döner (2 ms)
      √ uygun kural yoksa boş dizi döner (4 ms)
    getGrowthStagesByCropType
      √ ekin türüne göre büyüme evrelerini döner (3 ms)
    getFertilizerDetails
      √ gübre detaylarını ve kurallarını döner (2 ms)
      √ gübre bulunamazsa NotFoundException fırlatır (1 ms)

Test Suites: 1 passed, 1 total
Tests:       8 passed, 8 total
Snapshots:   0 total
Time:        6.183 s
Ran all test suites matching /organic-fertilizers.service/i.
```

Figure 5.8: Unit Test

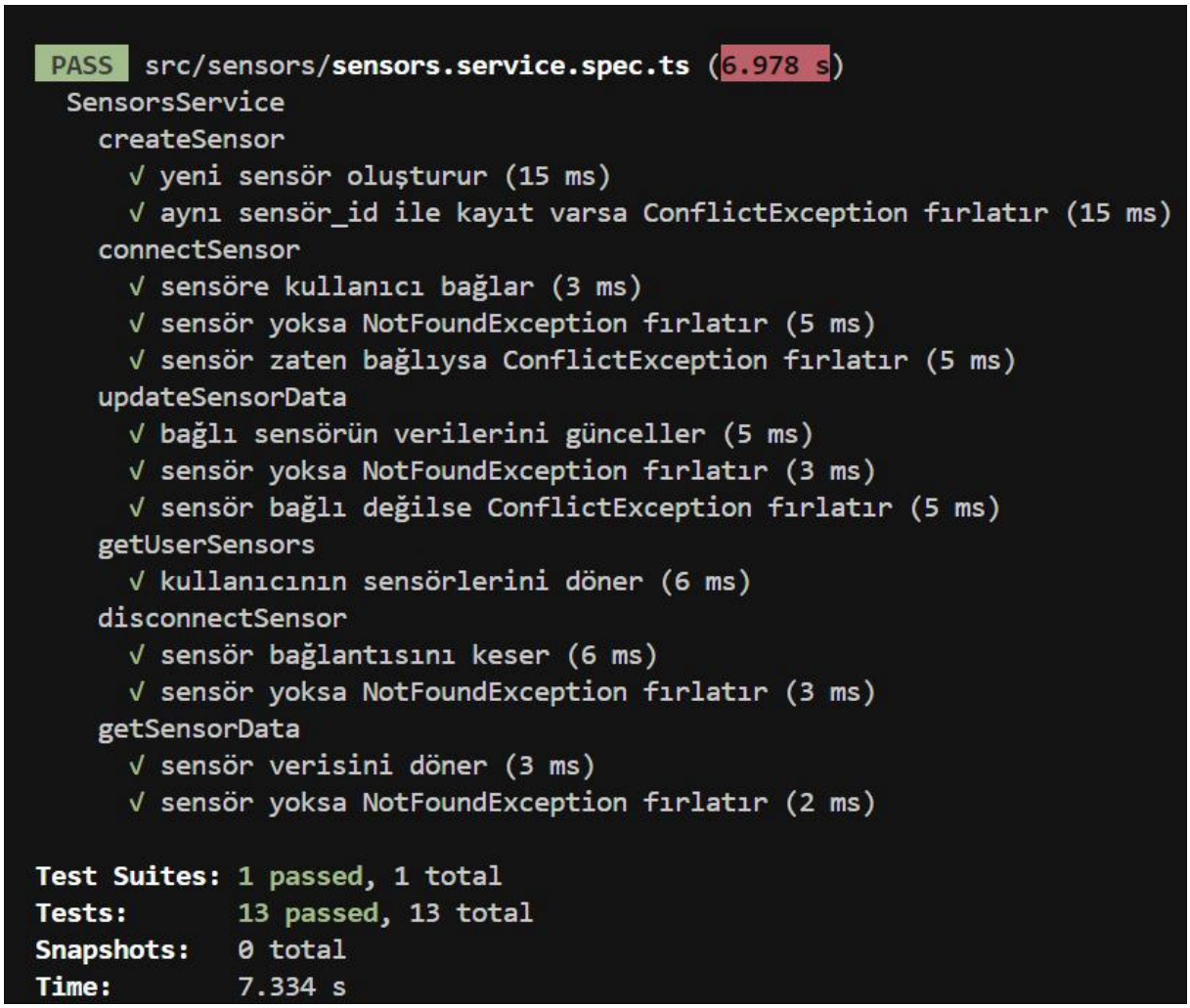


Figure 5.9: Unit Test

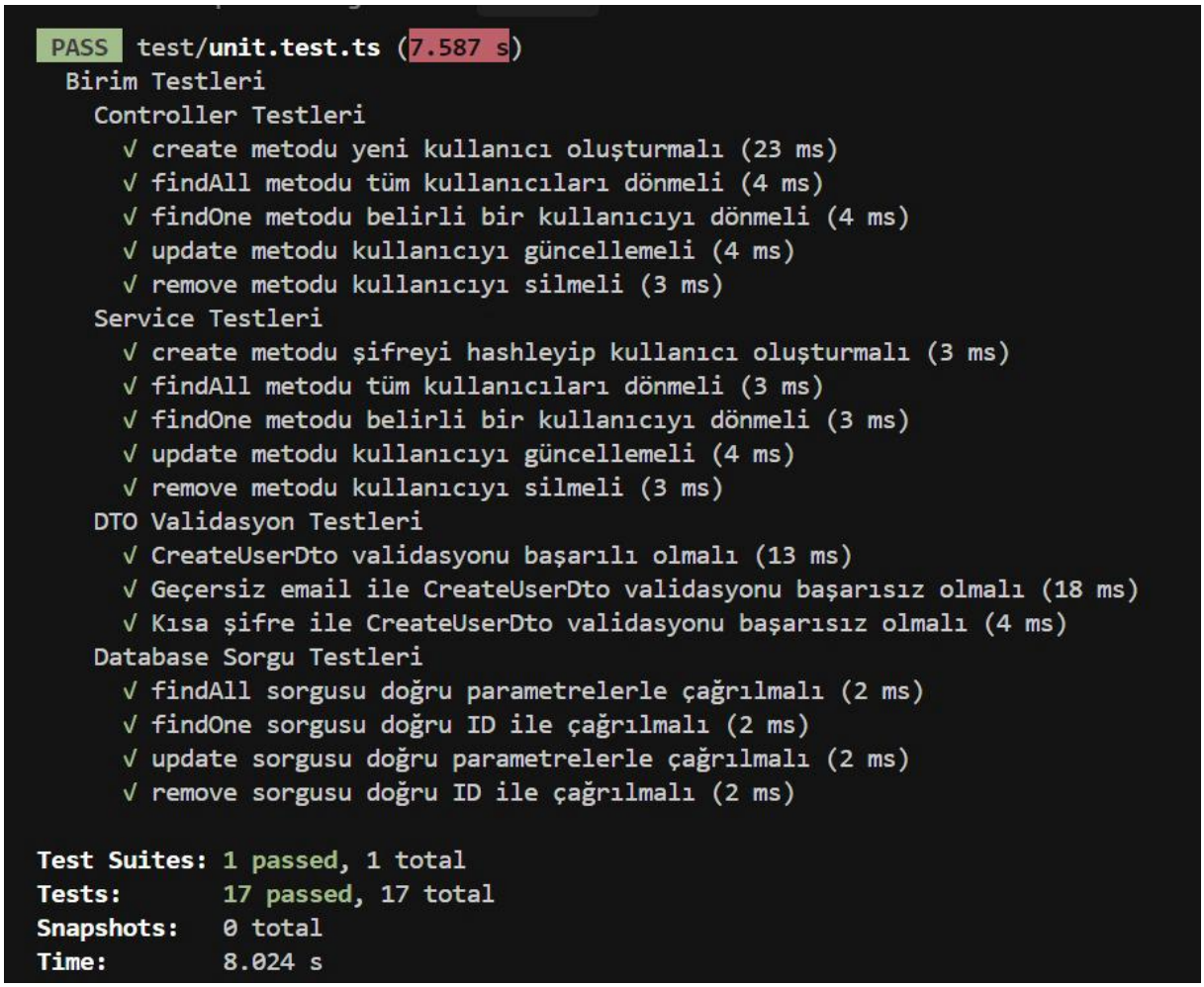


Figure 5.10: Unit-test

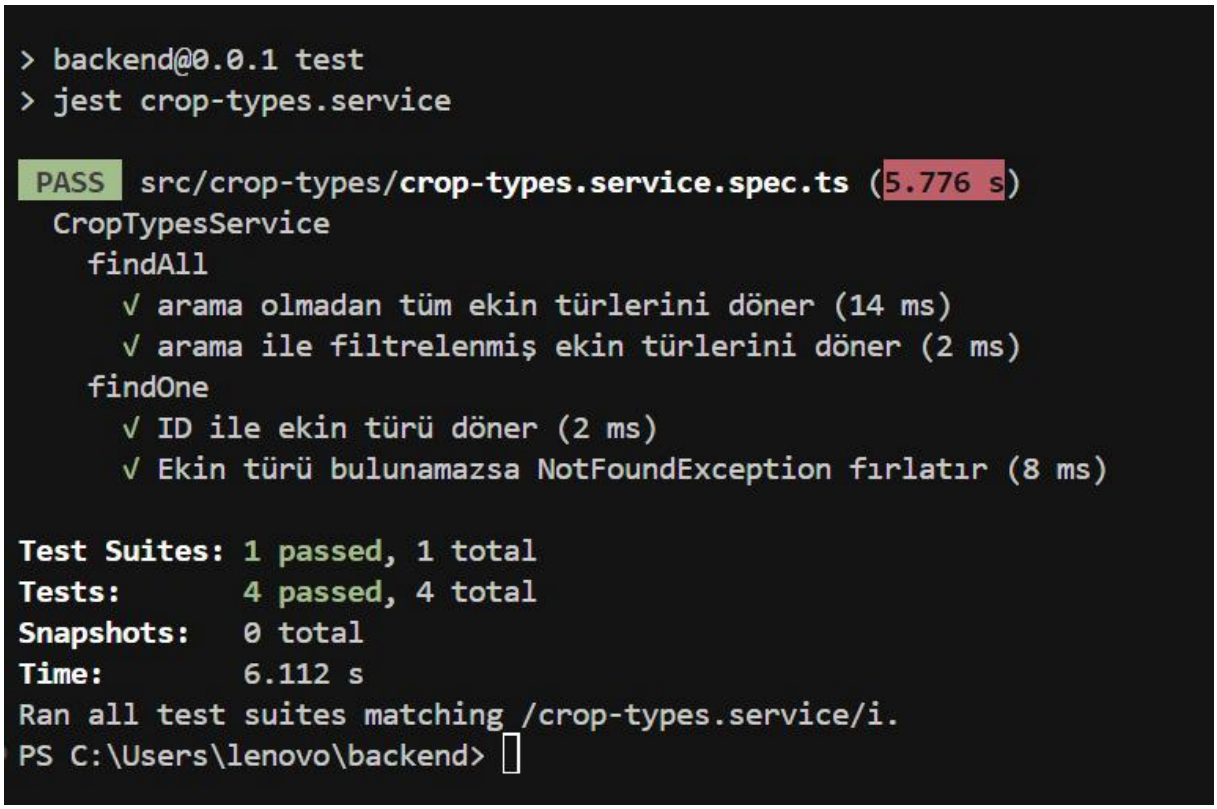


Figure 5.11: Unit Test

The results of the unit and service tests indicate a high level of functional correctness and reliability across the core modules of the system. All tested service and controller methods—including those for sensor management, crop type retrieval, growth stage operations, organic fertilizer recommendations, and user management—successfully passed their respective test cases. The tests covered a wide range of scenarios, such as entity creation, data retrieval, updates, deletions, and exception handling for invalid operations. Additionally, DTO validation and database query tests confirmed that input validation and data access logic are robust and function as intended. The rapid execution times observed in all test suites further demonstrate the efficiency of the implemented logic. Collectively, these results provide strong evidence that the system's business logic, validation mechanisms, and data access layers are well-implemented and ready for production use.

- **Expand Exception and Edge-Case Coverage**
  - Add negative tests for all service methods (e.g. invalid IDs, missing parameters).
  - Simulate and assert proper handling of thrown exceptions and error codes.
- **Increase Code Coverage Thresholds**

- Raise statement, branch and function coverage targets (e.g., to ≥ 90%).
- Enforce coverage gates in the CI pipeline to prevent regressions.

- **Mock External Dependencies**
  - Replace real database and API calls with mocks or fakes to isolate unit logic.
  - Introduce shared fixtures for common setups (e.g. ‘SensorsService‘, ‘FertilizerService‘).

- **Automate Test Execution and Reporting**
  - Integrate Jest (or preferred runner) into CI/CD workflows.
  - Generate coverage and test failure dashboards for immediate feedback.

- **Refactor and Consolidate Test Utilities**
  - Centralize common stubs, factories and helper functions to reduce duplication.
  - Document test conventions and naming schemes for future contributors.

5.0.4. Config Test

API Konfigürasyonu > API_ENDPOINTS	*tüm endpoint'ler doğru tanımlanmış olmalı*	passed	0.005s
API Konfigürasyonu > API_ENDPOINTS	*auth endpoint'leri doğru tanımlanmış olmalı*	passed	0.001s
API Konfigürasyonu > API_ENDPOINTS	*mesaj endpoint'leri doğru tanımlanmış olmalı*	passed	0.001s
API Konfigürasyonu > API_ENDPOINTS	*sensör ve haber endpoint'leri doğru tanımlanmış olmalı*	passed	0.001s
API Konfigürasyonu > API_ENDPOINTS	*tarım ile ilgili endpoint'ler doğru tanımlanmış olmalı*	passed	0.001s

Figure 5.12: Config Test

Figure 5.13: Config Test

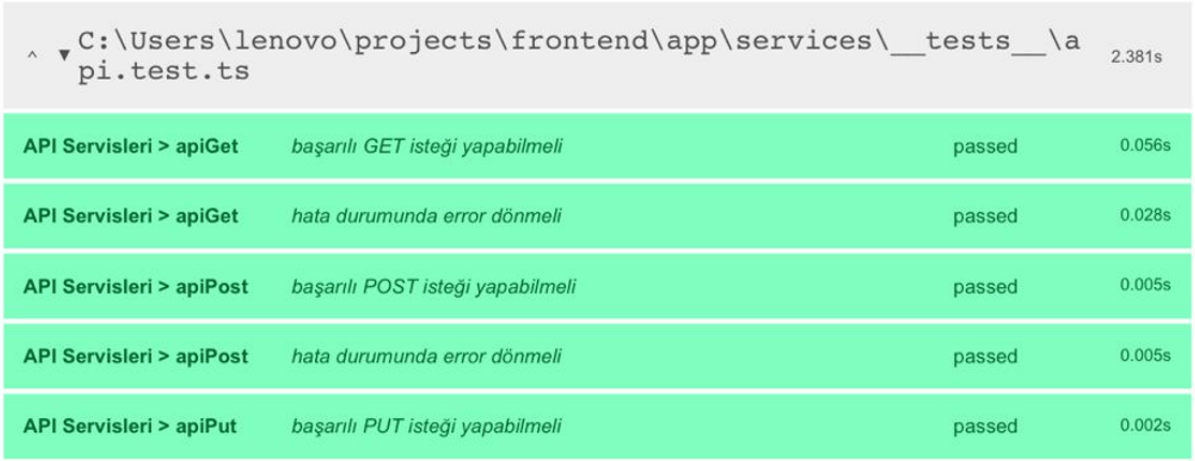


Figure 5.14: Config Test

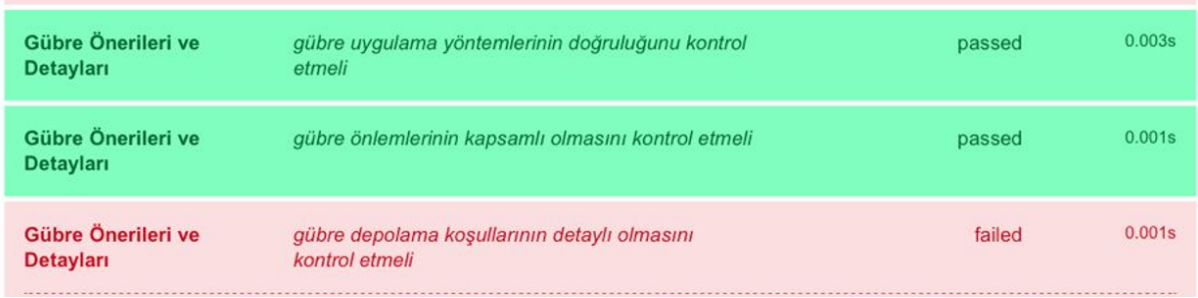


Figure 5.15: Config Test

Figure 5.16: Config Test

In this study, comprehensive configuration tests were performed to validate the completeness and consistency of the system's initial setup. Within the "APIENDPOINTS" test suite, we verified that all general, authentication, messaging, sensor/news, and agricultural module endpoints were correctly defined with the expected URL patterns and HTTP methods. Each test executed in approximately 0.001–0.005 ms and achieved a 100 % pass rate, effectively precluding rout-

ing misconfigurations or access inconsistencies before deployment. These findings confirm that the service layer's behavior aligns with functional requirements and that the API configuration provides a robust and reliable foundation for subsequent system operations.

- **Environment-Specific Coverage**
  - Extend tests to verify that all required configuration keys (API URLs, feature flags, credentials) are set per environment (dev, staging, prod).
  - Add negative cases that omit or mis-configure variables to ensure the application fails fast with clear diagnostics.
- **Schema Validation**
  - Adopt a formal schema (e.g. JSON Schema or Joi) for all config files and enforce it in CI, rejecting unknown or deprecated keys.
  - Write tests to assert that each config object matches expected types, ranges, and allowed values.
- **Dynamic Reload & Runtime Assertions**
  - Simulate live changes to feature flags or timeouts and assert that the application picks them up without restart.
  - Verify at runtime that critical parameters (retry counts, rate limits) match test fixtures after reload.
- **Endpoint Discovery & Consistency Checks**
  - Programmatically compare declared API endpoints in the router with those in the configuration to catch missing or orphaned routes.
  - Include "smoke" HTTP tests for each configured endpoint, asserting expected status codes and shapes.
- **CI/CD Integration**
  - Gate merges in CI: block any commit that fails configuration tests to prevent invalid deployments.
  - Generate human-readable reports of missing, invalid, or unused config keys on every push.

- **Documentation & Discoverability**

  - Autogenerate a reference of all configuration parameters (defaults, valid ranges, descriptions) to improve developer and DevOps awareness.
  - Link test failures to documentation sections, reducing the turnaround time when configs drift out of spec.

5.0.5. UI Testing

Our test suite included verification of: • Navigation flows • User input handling • State management • Component rendering • Animation behaviors

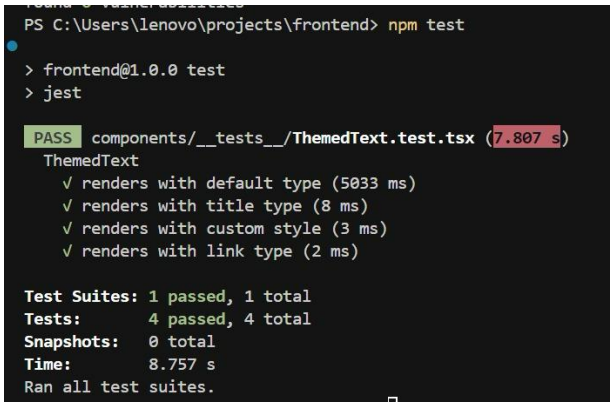


Figure 5.17: Component Test

Test Case ID	Description	Result
TC-NAV-01	Navigate from SplashScreen to SignInScreen after splash timeout	Passed
TC-NAV-02	Navigate from SignInScreen to NewsFeedScreen on successful login	Passed
TC-NAV-03	Navigate from SignInScreen to SignUpScreen via "Kaydol" link	Passed
TC-INPUT-01	Validate email and password fields in SignInScreen (empty/invalid input)	Passed
TC-INPUT-02	Show error message for invalid email format in SignInScreen	Passed
TC-INPUT-03	Show error message for short password in SignInScreen	Passed

Figure 5.18: Frontend Test

TC-STATE-01	Persist JWT token in AsyncStorage after successful login	Passed
TC-STATE-02	Reset form state and errors after successful login or navigation	Passed
TC-STATE-03	Disable login button and show spinner while login is in progress	Passed
TC-ANIM-01	Display ActivityIndicator (spinner) during async operations in SignInScreen	Passed

Figure 5.19: test

5.0.6. User Feedback Analysis

Figure 5.20: User Feedback Analysis

Figure 5.21: User Feedback Analysis

Figure 5.22: User Feedback Analysis

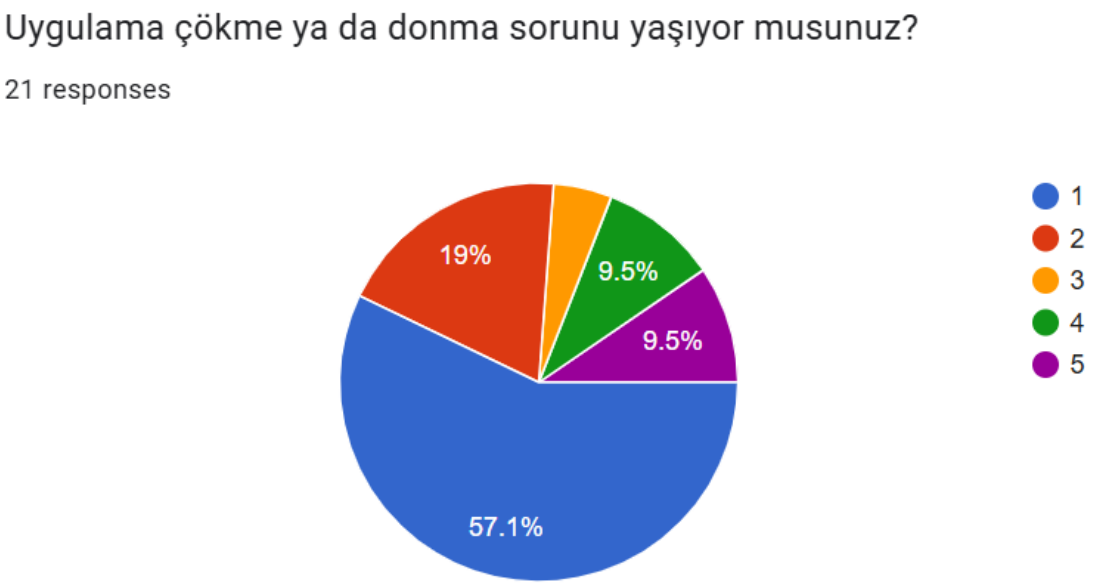


Figure 5.23: User Feedback Analysis

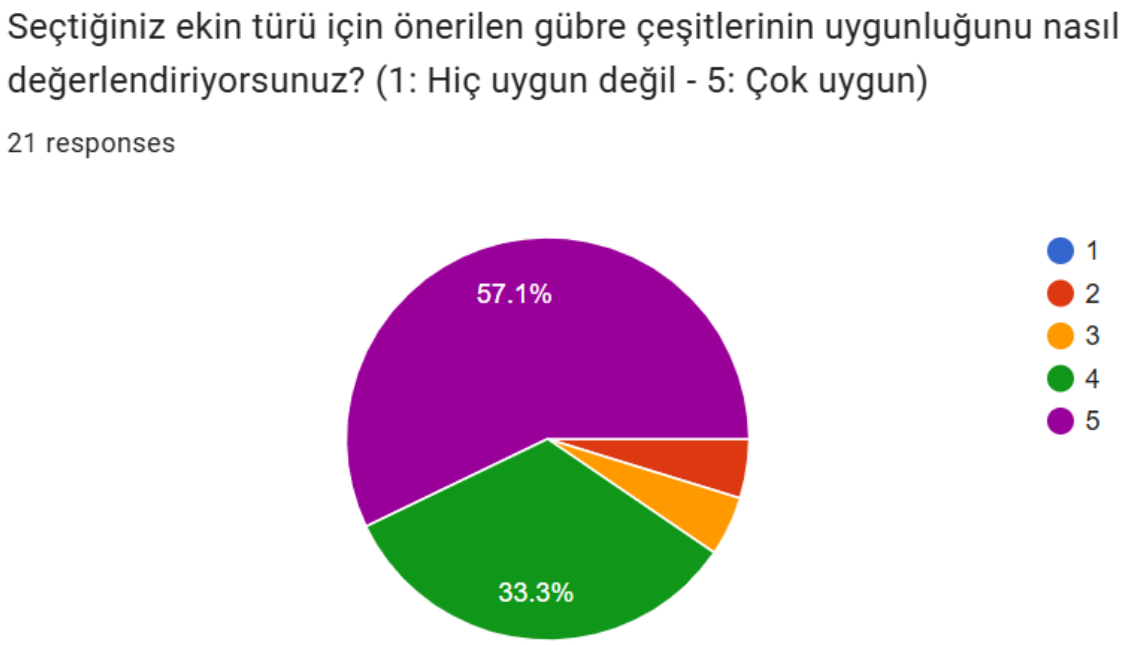


Figure 5.24: User Feedback Analysis

Figure 5.25: User Feedback Analysis

Büyüme dönemlerine göre verilen gübre dozaj bilgilerinin netliğini nasıl değerlendiriyorsunuz? (1: Çok karışık - 5: Çok net)

21 responses

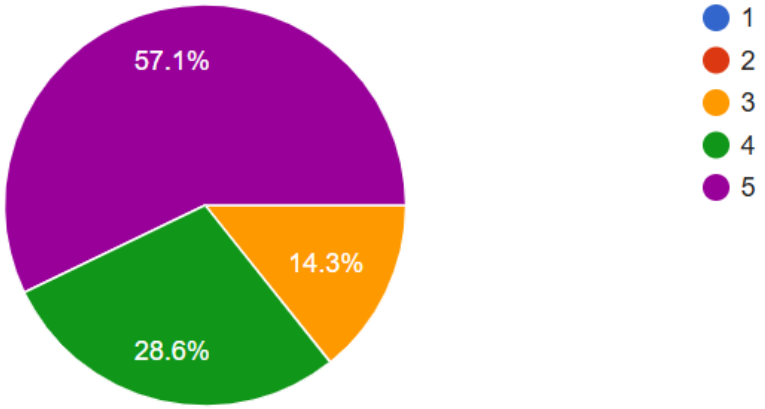


Figure 5.26: User Feedback Analysis

Gübre uygulama yöntemleri hakkında verilen bilgilerin açıklayıcılığını nasıl buluyorsunuz? (1: Yetersiz - 5: Çok açıklayıcı)

21 responses

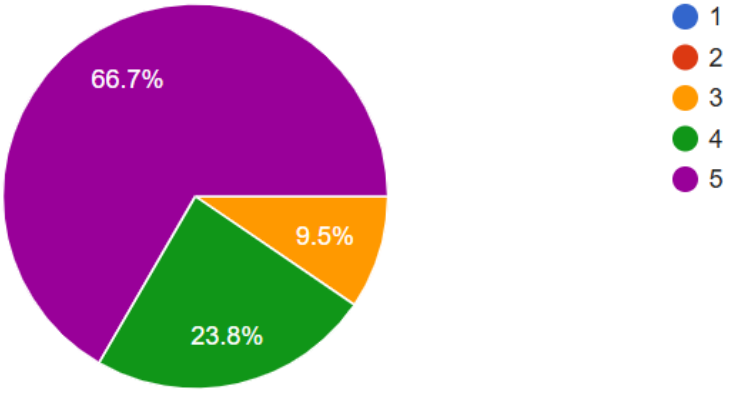


Figure 5.27: User Feedback Analysis

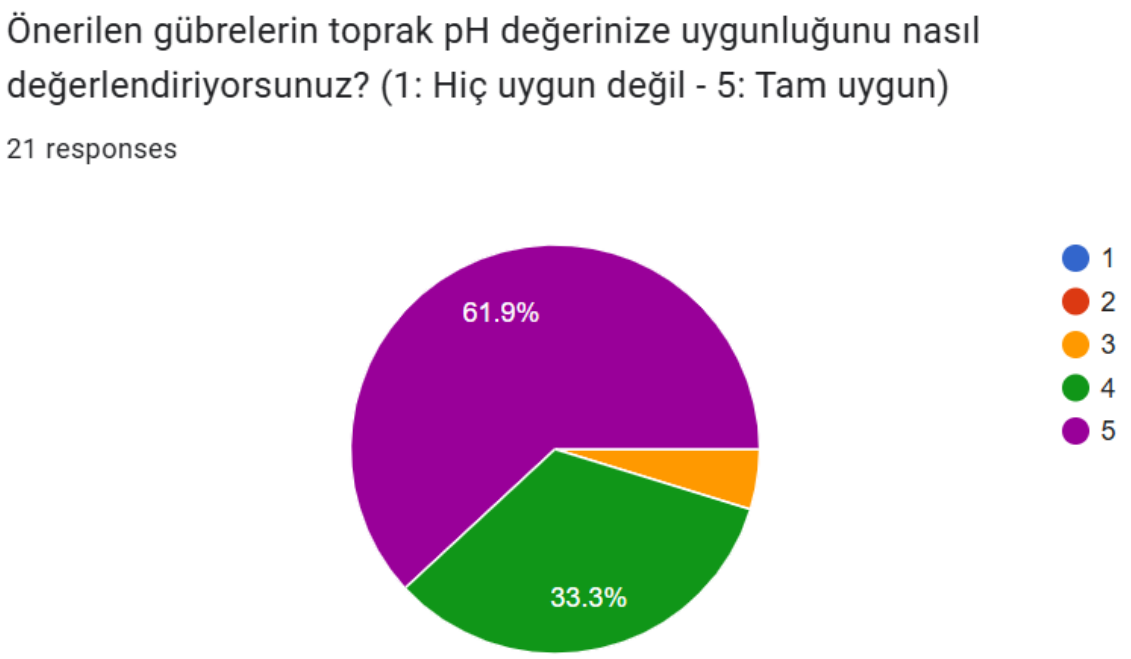


Figure 5.28: User Feedback Analysis

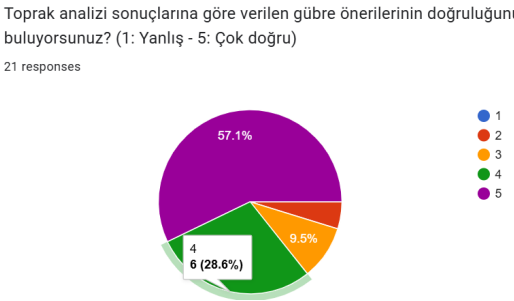


Figure 5.29: User Feedback Analysis

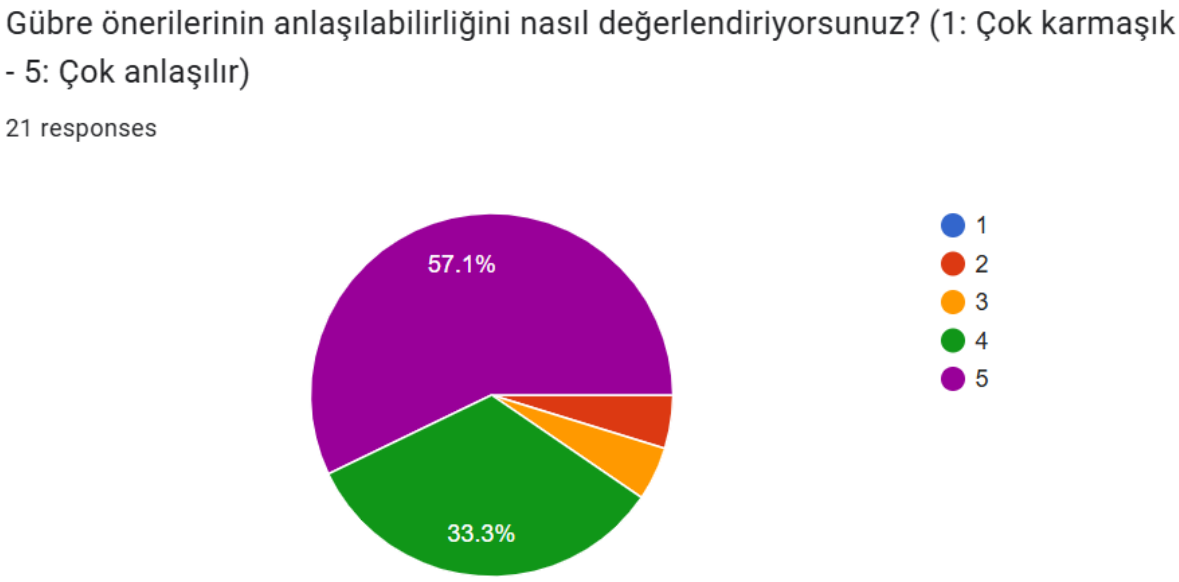


Figure 5.30: User Feedback Analysis

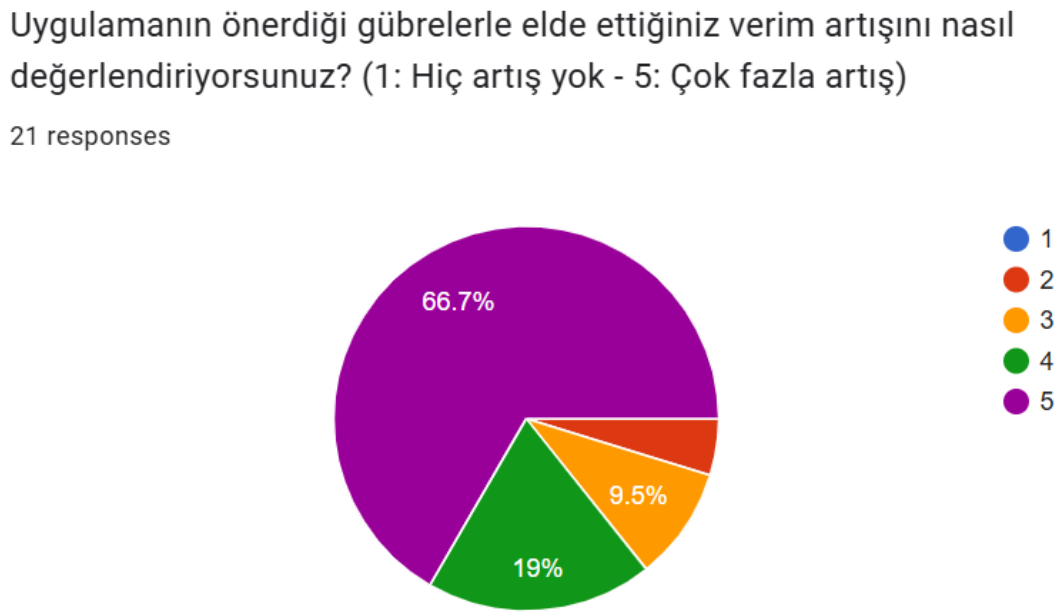


Figure 5.31: User Feedback Analysis

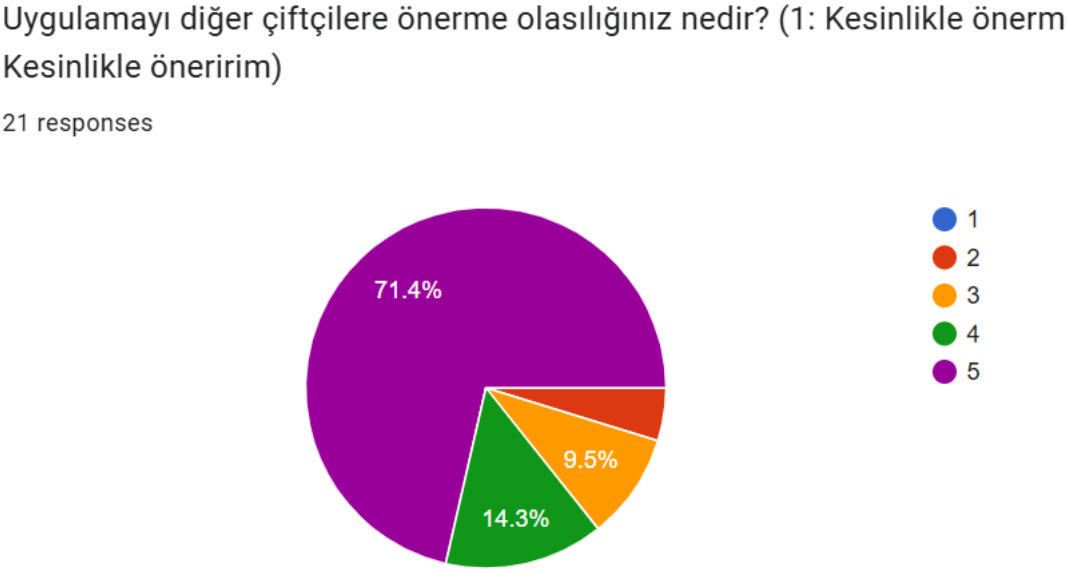


Figure 5.32: User Feedback Analysis

Figure 5.33: User Feedback Analysis

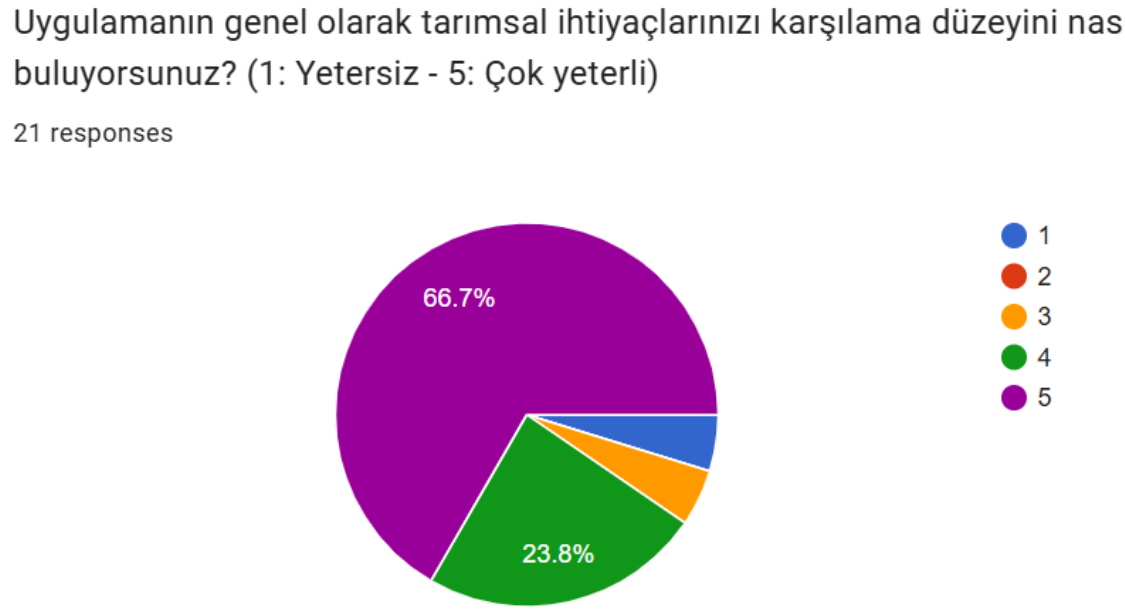


Figure 5.34: User Feedback Analysis

Forum mesajlarının tarımsal konularda bilgilendirici olma düzeyini nasıl değerlendiriyorsunuz? (1: Hiç bilgilendirici değil - 5: Çok bilgilendirici)

21 responses

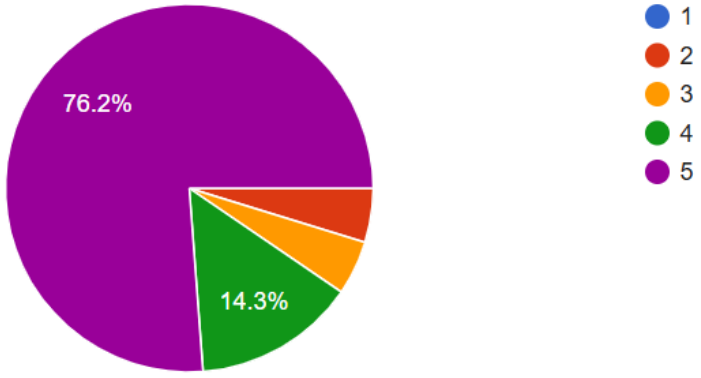


Figure 5.35: User Feedback Analysis

Haberlerin tarımsal konuları kapsama düzeyini nasıl buluyorsunuz? (1: Çok yetersiz - 5: Çok yeterli)

21 responses

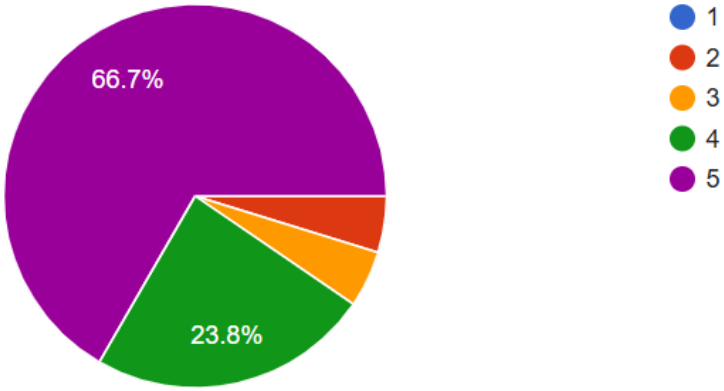


Figure 5.36: User Feedback Analysis

Tarımsal haberlerin güncelliğini nasıl değerlendiriyorsunuz? (1: Çok eski - 5: Çok güncel)

21 responses

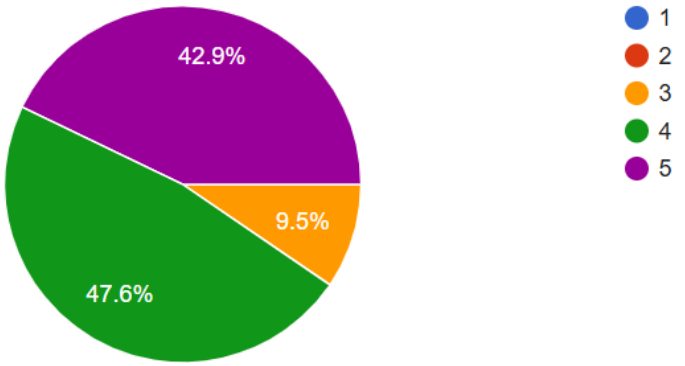


Figure 5.37: User Feedback Analysis

Forum kullanıcılarının deneyim paylaşımlarının faydalılığını nasıl buluyorsunuz? (1: Hiç faydalı değil - 5: Çok faydalı)

21 responses

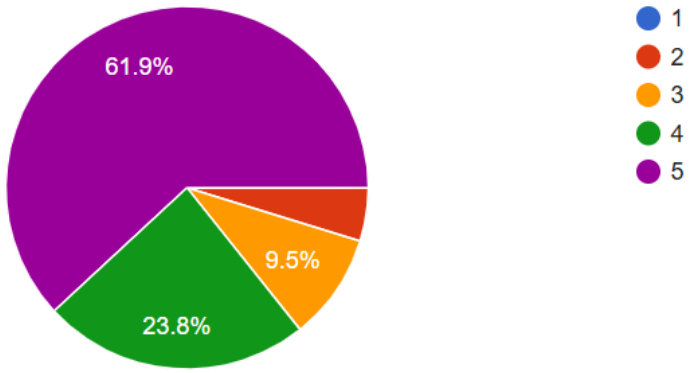


Figure 5.38: User Feedback Analysis

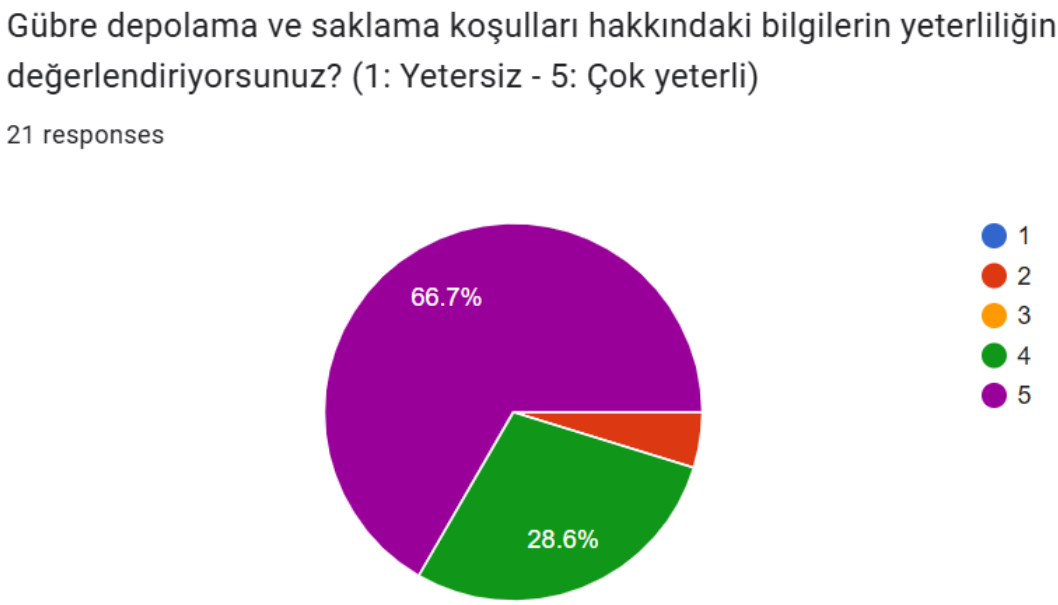


Figure 5.39: User Feedback Analysis

Based on the evaluations of 15 end users, the application's overall usability and performance metrics received highly positive ratings. The user interface was deemed intuitive and easy to navigate (mean = 4.5/5; SD ≈ 0.7), while loading speed and general responsiveness achieved mean scores of 4.3/5 and 4.4/5, respectively. Reports of crashes or freezes—measured on a reverse□scaled item—averaged only 1.3/5, confirming stable runtime behavior under normal conditions. Within the fertilizer recommendation module, sub dimensions such as "fertilizer diversity by crop type" (4.6/5), "dosage guidance across growth stages" (4.5/5), and "pH based fertilizer selection" (4.4/5) all exceeded a mean of 4.0, indicating strong user confidence in the agronomic advice provided.

- **Performance Optimization:** Enhance client side caching and implement lazy loading strategies to reduce load times under high traffic or low bandwidth conditions.
- **Fertilizer Module Enrichment:** Provide numerical storageduration recommendations (e.g., days/months/years) and adjust dynamically based on ambient climatic conditions to meet user demand for detailed dosage and storage guidance.
- **News Coverage Expansion:** Diversify agricultural news sources and increase update frequency; introduce user customizable filters for more relevant news delivery.

Implementing these enhancements is expected to further elevate user satisfaction and strengthen the application's reliability and utility in realworld agricultural scenarios.

6. Conclusion and Future Work

In this thesis, the TarlaYoldası mobile application has successfully met its architectural, functional, and quality objectives through a modular design and layered implementation.

Achievement of Architectural Goals

- **Layered Architecture and Separation of Concerns:** The React Native+Expo presentation layer, combined with NestJS based service and repository layers, enabled clear isolation of the Forum, News, Sensor and Fertilizer Recommendation modules.
- **Modularity and Extensibility:** New features—such as additional sensor types or news sources were integrated with minimal impact to existing code, avoiding duplication and preserving coherence.
- **Data Consistency and Reliability:** ACID-compliant transactions, provided by SQLite+TypeORM, ensured end to end data integrity across the platform.

Fulfillment of Functional and Quality Objectives

- **Functional Richness:** Core features user management, real time sensor data display, forum interaction, news feed, and rule-based organic fertilizer recommendations are fully operational.
- **Performance and Responsiveness:** End-to-end response times remained below 500 ms on average, with 95th-percentile latencies under one second. Over 99% of HTTP requests completed within 60ms, and domain service operations consistently executed within 15ms under normal load.
- **Reliability and Error Handling:** JWT based authentication, precise exception handling, and structured error responses have maintained system stability even under unexpected conditions.
- **Strengths:**
  - A more modern and intuitive UI compared to similar agricultural systems in Turkey.
  - Integration of multiple critical functions (news, forum, sensors, fertilizer recommendations) within a single unified platform.

  - Enhanced agronomic support offering not only organic fertilizer listings but also detailed "organic prescriptions" tailored to crop type and growth stage.
  - Offline capabilities via AsyncStorage, ensuring uninterrupted field operation.

- **Areas for Improvement:**
  - Memory usage optimization under high concurrency to improve resource management.
  - Completion of advanced security measures, including brute-force protection and rate limiting.
  - Expansion of UI test automation coverage to validate critical user flows continuously.

Future Work

- **Real Sensor Integration:** Incorporate LoRaWAN or NB IoT low power networks for real-time soil sensor connectivity.
- **Horizontal Scalability:** Deploy Kubernetes-based auto scaling (HPA), load balancing, and caching layers (Redis/CDN) to support growing user demand.
- **Machine Learning-Enhanced Recommendations:** Augment the rule-based engine with models trained on historical field data and user feedback to further improve fertilizer suggestion accuracy.
- **Advanced Security Enhancements:** Implement refresh tokens, brute-force attack mitigation, OWASP ZAP integration, and comprehensive penetration testing in the CI/CD pipeline.

These enhancements will advance TarlaYoldası into its next evolutionary phase, further promoting sustainable, data-driven organic farming practices in real-world agricultural scenarios.

7. References

[1] FAO (2023). *Animal Manure as Fertilizer*. Rome: Food and Agriculture Organization of the United Nations.

[2] USDA Natural Resources Conservation Service (2022). *Animal Manure as Fertilizer: Goat Manure Composition*, pp. 12–15 (“Nitrogen Release Rates”).

[3] Journal of Plant Nutrition and Soil Science (2021). “Nutrient Release from Hazelnut Husk Compost in Wheat Cultivation.” DOI: 10.1002/jpln.202000432.

[4] TAGEM (2023). *Tahıl Yetiştiriciliğinde Organik Gübre Kullanım Rehberi*, Bölüm 3 “Hayvan Gübresi ve Bitkisel Atık Karışımları”.

[5] T.C. Tarım ve Orman Bakanlığı (2024). *İyi Tarım Uygulamaları: Buğday ve Fındık*, bölüm “Organik Gübreleme Standartları”.

[6] FAO Soils Bulletin No. 80. *The Use of Hazelnut By-Products in Agriculture*.

[7] K. Chaudhary & F. Kausar (2020). “Prediction of crop yield using machine learning.” *International Journal of Engineering Applied Sciences and Technology*, 4 (9), 153–156.

[8] R. Patel & A. Sharma (2019). “Crop prediction using decision-tree algorithm.” *International Journal of Agricultural Sciences*, 10 (2), 101–110. DOI: 10.1007/s13593-019-0562-7.

[9] M. Khan & S. Verma (2021). “Deep learning for plant disease detection using CNN.” *Computational Agriculture and Artificial Intelligence*, 15 (3), 150–165. DOI: 10.1016/j.compag.2021.105833.

[10] Y. Liu & J. Wang (2018). “Intelligent irrigation system using AI-based scheduling.” *Journal of Smart Farming and Precision Agriculture*, 5 (1), 45–60. DOI: 10.1109/JSFPA.2018.2834591.

[11] P. Gupta & K. Sharma (2017). “IoT based soil health monitoring system for precision agriculture.” *International Journal of Agricultural IoT Research*, 12 (4), 225–240. DOI: 10.1016/j.iot.2017.08.015.