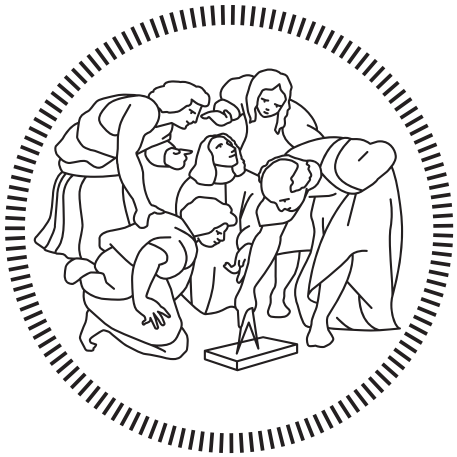

POLITECNICO DI MILANO

Computer Science and Engineering

# Design Document

## DREAM - Data-dRiven PrEdictive FArMing in Telangana

Software Engineering 2 Project

Academic year 2021 - 2022

January 9, 2022

Version 1.0

*Authors*:
Kinga Marek
Józef Piechaczek
Mariusz Wiśniewski

*Professor*:
Damian Andrew Tamburri

# Contents

# Chapter 1

# Introduction

## 1.1 Purpose

The goal of this document is to offer broad advice to the architectural design of the software product, hence it is primarily directed to the development team, which includes both developers and testers.

More specifically, this document provides an overview of the high-level architecture (identifying the key components and their interactions) as well as further information on the application's runtime behavior and user interface. Additionally, it comprises a strategy for implementation, integration, and testing.

This document, together with RASD [1], is intended to aid developers in the implementation of the DREAM project.

### 1.1.1 Goals

**G1.** Improve farmers performance by providing them with personalized suggestions.

**G2.** Acquire, combine, and visualize data from external systems.

**G3.** Facilitate performance assessment of the farmers.

**G4.** Promote regular farms' visits by agronomists, depending on the type of problems they face.

**G5.** Enable agronomists to exchange information with farmers.

**G6.** Enable farmers to exchange their knowledge.

## 1.2 Scope

Data dRiven PrEdictive FArMing (DREAM) is a project that aims to restructure the food production process in Telangana in order to build more resilient agricultural systems needed to fulfill the region's expanding food demand [2]. To handle such a large issue, it is necessary to bring together people from many professions and backgrounds; hence, participation from groups such as stakeholders, policymakers, farmers, analysts, and agronomists is sought.

Data collection and analysis can be considered as the first possible step in achieving the goal. Farmers, specialized sensors planted on the ground that monitor soil humidity, and government agronomists who visit farms on a regular basis might all provide this information.

The system's objective is to support policymakers to identify farmers who perform well, particularly in harsh weather circumstances, farmers who perform poorly and require support, and to determine if agronomist-led steering efforts provide significant results. This will be accomplished by continual evaluation based on data visualizations.

Farmers would profit from the system as well, because it would allow them to share best practices between each other and request assistance when required. All of this will be augmented with data visualizations, such as suggestions and weather forecasts.

The initiative also promises to make agronomists' jobs easier by generating a daily schedule for field visits, providing data on farmers' performance, and telling them about incoming support requests.

The system aims to solve a very challenging problem, which involves the collaboration of many professionals. All of these characteristics necessitate that the system is straightforward to use, intuitive, and plain so that everyone may benefit from it.

## 1.3 Definitions, Acronyms, Abbreviations

### 1.3.1 Definitions

| Expression | Definition |
|---|---|
| Mockup | A static wireframe that adds additional aesthetic and visual user interface aspects to provide a realistic depiction of what the final page will look like. |
| Water irrigation system | Collection of external devices which gather information about the amount of water used on a given farm. Collected data are available via an API. |
| Sensor/Humidity sensor | External device which collects information about the humidity of soil on a given farm. |
| Sensor system | Collection of all the sensors on a farm. Collected data are available via an API. |

| | |
|---|---|
| Farmer's summary | Summary of farmer and farm data. Contains: information about farmer and his farm, note history, short-term weather forecasts, production data, farm visits, weather forecast history, soil humidity data, water usage, and help requests. |
| Farmer's note | Evaluation note given by a governmental policy maker based on a farmer's performance. It can include a type of problem, that the farmer is facing. Can be negative, neutral, or positive. |
| Farm data | Data concerning a farm, inserted by a farmer during account creation. Consist of water irrigation system's ID, sensor system's ID and farm's address (address line 1, address line 2, postal code, city and mandal). |
| Personalized suggestion | Suggestions proposed to the farmers, based on a mandal and types of production of a given farm. Suggestions are inserted into the system by agronomists. |
| Production type | Kind of plants being produced on Telangana's farms. |
| Production data | Information provided by a farmer about his production. It lists production types and quantities (in kg) generated per type. |
| Mandal | A local government area in India. Part of a district, which in turn is a part of a state. |
| Dashboard | The first screen that appears to the user after logging in to the system. |
| Area of responsibility | Set of mandals that an agronomist is responsible for. |
| Daily plan | A daily schedule for visiting farms in a certain area of responsibility. |
| Casual visit | A visit automatically arranged by the system. There are two casual visits to each farm every year. |
| Request for help, Help request | An issue created by a farmer. It contains a topic, description and a problem. The system automatically forwards it to well-performing farmers and agronomists in the same mandal. |
| External systems | Systems that are not a part of DREAM and collect various farm-specific data and share them via an API. Following external systems are used: Weather Forecast System, Water Irrigation System, Sensor System. |
| Water irrigation system's ID | Number which uniquely identifies a water irrigation system. It is necessary for definition of the external system's API endpoint. The number is provided during installation of the water irrigation system. |
| Sensor system's ID | Number which uniquely identifies a sensor system. It is necessary for definition of the external system's API endpoint. The number is provided during installation of the sensor system. |

### 1.3.2 Acronyms

| Acronyms | Expression |
|---|---|
| A | Domain assumption |
| API | Application Programming Interface |
| D | Dependency |
| DB | Database |
| DD | Design Document |
| DREAM | Data-dRiven PrEdictive FArMing in Telangana |
| G | Goal |
| HMACSHA1 | Hash-based Message Authentication Code (HMAC) using the SHA1 hash function |
| HTTP | Hypertext Transfer Protocol |
| JSON | JavaScript Object Notation |
| JWT | JSON Web Token |
| M | Mockup |
| MVP | Minimum Viable Product |
| PBKDF2 | Password-Based Key Derivation Function 2 |
| R | Requirement |
| RASD | Requirements Analysis and Specifications Document |
| REST | Representational State Transfer |
| SQL | Structured Query Language |
| SMTP | Simple Mail Transfer Protocol |
| UI | User Interface |

### 1.3.3 Abbreviations

| Abbreviations | Expression |
|---|---|
| i.e. | id est |

## 1.4 Revision History

| Date | Revision | Notes |
|---|---|---|
| | v.1.0 | First release. |

## 1.5 Reference Documents

[1] Kinga Marek, Józef Piechaczek, and Mariusz Wiśniewski. *Requirements Analysis and Specifications Document, DREAM - Data-dRiven PrEdictive FArMing in Telangana*. Dec. 2021.

[2] Elisabetta Di Nitto, Matteo Rossi, and Damian Tamburri. *A.Y. 2021-2022 Software Engineering 2 Requirement Engineering and Design Project: goal, schedule, and rules*. Oct. 2021.

## 1.6 Document Structure

1. **Introduction**: outlines the document's overall purpose as well as its scope. It also establishes particular definitions, acronyms, and abbreviations that are used throughout the text.
2. **Architectural Design**: provides a high-level overview of how the system is organized into components and identifies interactions between them, as well as component, deployment, and runtime views. This section also focuses on the key architectural styles and patterns used in the system's design.
3. **User Interface Design**: gives an overview of how the system's user interface would appear and defines the system's functionality from the user's point of view.
4. **Requirements Traceability**: describes how the requirements specified in the RASD translate to the design components defined in this document.
5. **Implementation, Integration and Test Plan**: specifies the order in which the system's subcomponents are to be implemented and then integrated. Further, it provides a plan to follow in order to properly test the integration of system components.
6. **Effort Spent**: provides details on each group member's contribution to the project.
7. **References**

# Chapter 2

# Architectural Design

## 2.1 Overview

At the high level, the system employs a client-server architecture. The client part consists of the web application running in a browser on a user's machine, whereas the server part is composed of web, DREAM, mail, and SQL servers. The high-level architecture is presented in the figure 2.1.

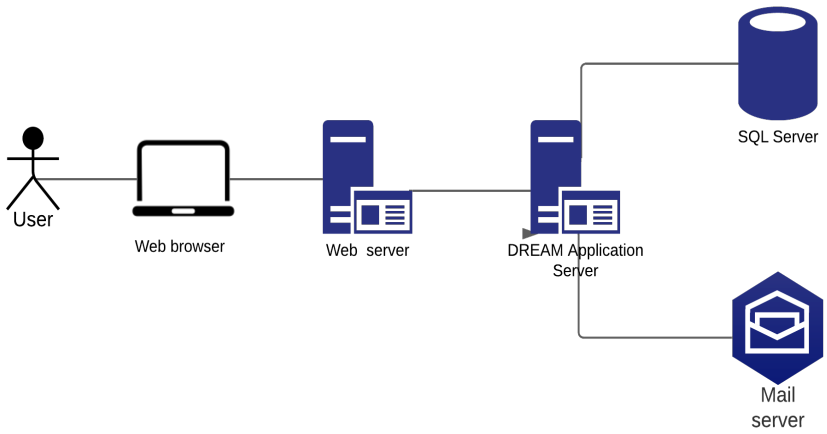


Figure 2.1: High-level architecture diagram

The DREAM server part uses a database hosted on a SQL Server to store data collected from the users as well as external systems, and the Mail Server used to send the emails required for password reset. Moreover, the web server works as a middleware between the user interface and the database, utilizing HTTP protocol to receive and respond to requests from the web application running on a client's web browser. Finally, the *DREAM server* hosts the application server (*backend* part) with the majority of the application business logic.

## 2.2 Component View

### 2.2.1 High level

The following diagram presents a high level overview of the whole DREAM system. It describes both structure of the system and high level interaction within it. The description of the system components, as well as the low level diagram of the application server, are presented below.

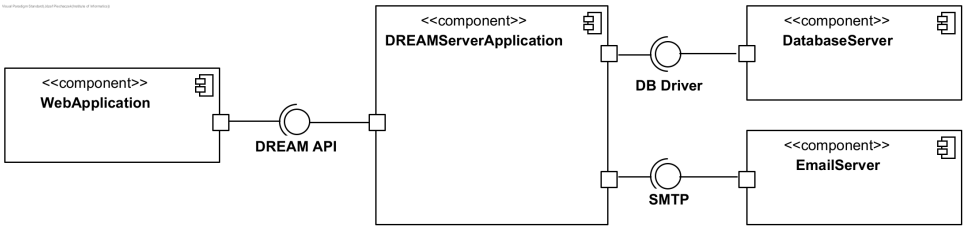


Figure 2.2: DREAM Application high level component diagram

The system consists of the following components:

- **WebApplication** - frontend part of the application, reachable via any modern browser, regardless of the form factor of the device.
- **DREAMServerApplication** - backend part of the application, handling main application login. It provides a REST API [11] interface, which is consumed by the WebServer. This component is described in detail in further parts of the document.
- **DatabaseServer** - part of the application responsible for data storage and management. Communicates with DreamServer via a DB Driver.
- **EmailServer** - additional service, allowing system to send a password reminder e-mail using SMTP.

### 2.2.2 DREAM server

Detailed component diagram of the DREAM Server is presented in the figure 2.3. The server application was broken down into 3 layers using N-tiers architecture pattern described in section 2.6. The communication between layers is established by implementation of interfaces presented in the diagram. In accordance with the N-tier pattern, the communication runs only in one direction – starting from the API layer down to the Data Access layer.

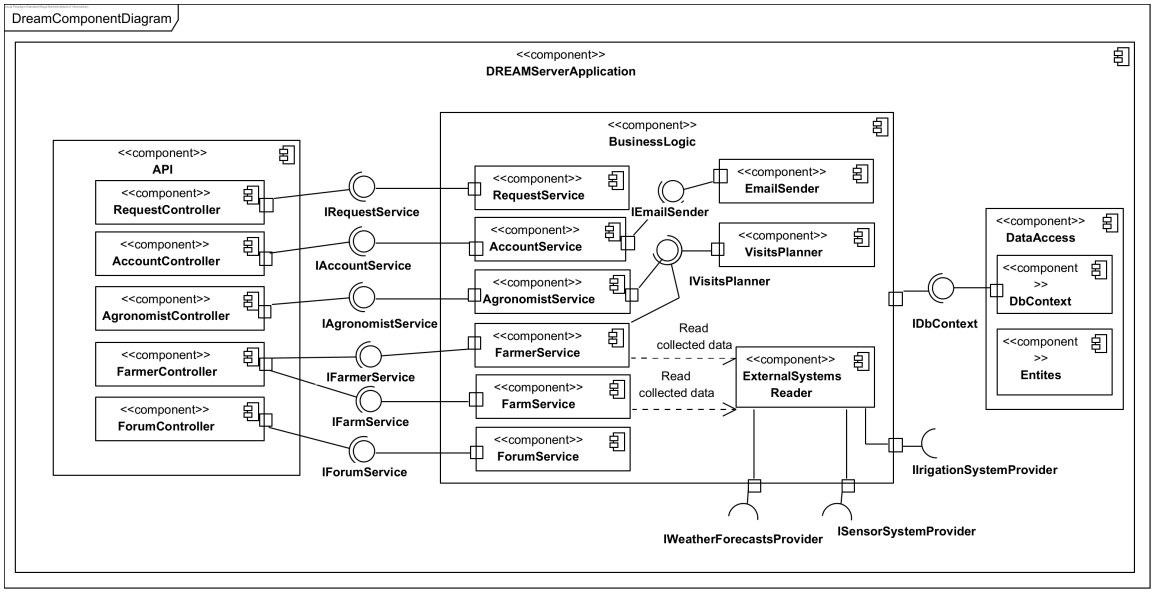


Figure 2.3: DREAM server component diagram

The first layer, the API layer, is composed by a set of controllers that describe available REST endpoints. Then, the middle layer, Business Logic layer contains services and other classes that provide logic necessary for handling HTTP requests from the client application and cooperation with external systems. In detail, the Business logic layer contains:

- **RequestService** - handles creating, editing, deleting and accessing help requests together with logic required for automatic selection of requests' recipients.
- **AccountService** - handles account registration and deletion, password reset and authentication logic.
- **AgronomistService** - handles area of responsibility and daily plan management, setting execution state of visits. It calls VisistsPlanner about the necessities to plan new visits after each submission of daily plan execution state or rejection of planned visit.
- **FarmerService** - handles notes assignment and providing data required for building farmer's summary (access to notes history). In addition, it allows searching for specified farmer using different query filters.
- **FarmService** - handles management of farmers' production data, providing data required for building farmer's summary (farm's external systems and production data).
- **ForumService** - handles accessing and creating forum threads; accessing, creating and deleting comments.
- **SuggestionService** - handles creating, deleting and reading suggestions by agronomists. Moreover, retrieves personalized suggestions for the farmer based on his farm's location and production type.
- **WeatherForecastService** - handles retrieving different weather forecasts data based on provided query parameters.

- **EmailSender** - handles logic required for e-mail sending during password reset process.
- **VisitsPlanner** - handles implementation of *visits scheduling algorithm* described in subsection 2.7.1.
- **ExternalSystemsReader** - handles logic required for execution of daily jobs that read and store in the database data obtained from IWeatherForecastsProvider, ISensorSystemProvider and IIrigationSystemProvider interfaces. The implementation of these interfaces is provided by the external systems' API.

Finally, the Data Access layer contains entities and classes required to efficiently communicate with the database. The components of this layer were not presented in detail, as the communication with the database will leverage object relational mapping, described in more detailed way in the component interfaces section 2.11.

## 2.3 Deployment View

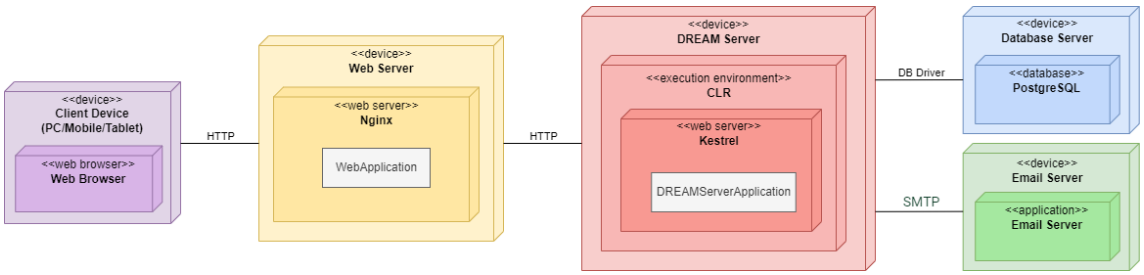


Figure 2.4: Deployment diagram

The system is divided into 5 parts: DREAM Server, Web Server, Database Server, Client Device, and Email Server.

- **DREAM Server** - DREAM backend server, it handles application logic and provides a REST API. It runs using Kestrel Web Server [15], which is loaded by Common Language Runtime (CLR) [14], the virtual machine component of Microsoft .NET Framework.
- **Web Server** - DREAM frontend server, it provides the user interface layer. It hosts the static version of the website using Nginx [16]. The website communicates with the Application Server tier via REST API using HTTP requests.
- **Database Server** - Database storage system, using PostgreSQL [12]. It allows data access using DB driver.
- **Client Device** - Device used by clients to access the website hosted in Web Server tier using HTTP requests. It can be any type of device, capable of running a modern web browser
- **Email Server** - Email server, allowing to send messages via SMTP protocol.

## 2.4 Runtime View

The sequence diagrams presented in this chapter illustrate the application's runtime perspective. They define how components (introduced in the section 2.2) interact with one another to realize the DREAM's most essential use cases in a more detailed way than it was depicted in RASD. The diagrams assume that requests sent to *DbContext* are always handled correctly (which cannot be guaranteed), so that their complexity does not expand dramatically.

### Create account

The ability to create a user account is the most essential functionality provided by the system. As already mentioned in RASD, DREAM provides three types of actors: agronomists, farmers, and policy makers. The registration data varies depending on the chosen role, i.e. every agronomist must specify his area of responsibility, whereas every farmer needs to provide data regarding his farm. All of this is handled via the *WebApplication* component. Subsequently, *AccountController* forwards the request to *AccountService*, which interacts directly with *DbContext*. Moreover, when a farmer creates his account, *AccountService* ensures that both a farm and two casual visits are created instantly, thus satisfying the requirement **R5**. The whole process is presented in the figure 2.5.

### Request help

The sequence diagram displayed in the figure 2.6 shows the process of creating a help request by the farmer. After specifying its topic and contents, *WebApplication* forwards that data to *RequestController*, which then interacts with *DbContext* via *RequestService*. Correctly created help request is displayed to the farmer afterwards.

### Assess farmer's performance

Giving policy makers the possibility to assess farmers' performance is one of the most important functionalities of DREAM. It takes a coordination of the following components to fulfill this use case: *WebApplication*, *FarmerController*, *FarmerService*, *RequestController*, *RequestService*, *VisitsPlanner*, and *DbContext*. Furthermore, supplementary collaboration of *FarmService*, *WeatherForecastController*, and *WeatherForecastService* is required in order to construct the farmer's summary. The diagram presented in the figure 2.7 provides a thorough description of this process. It leverages the visit scheduling algorithm (see section 2.7.1) to satisfy the requirements **R17** and **R18**.

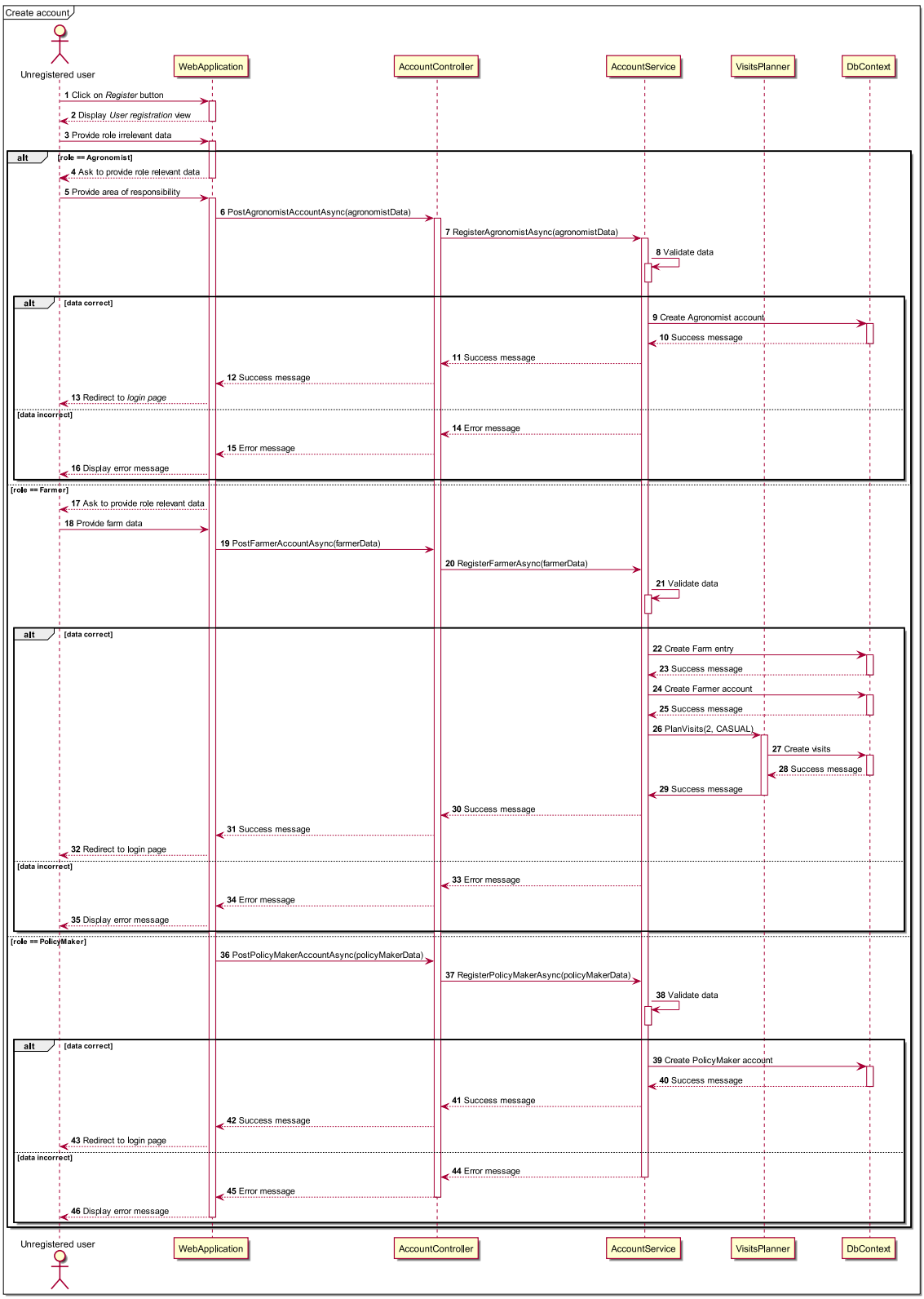


Figure 2.5: Sequence diagram presenting the process of creating a new user account.

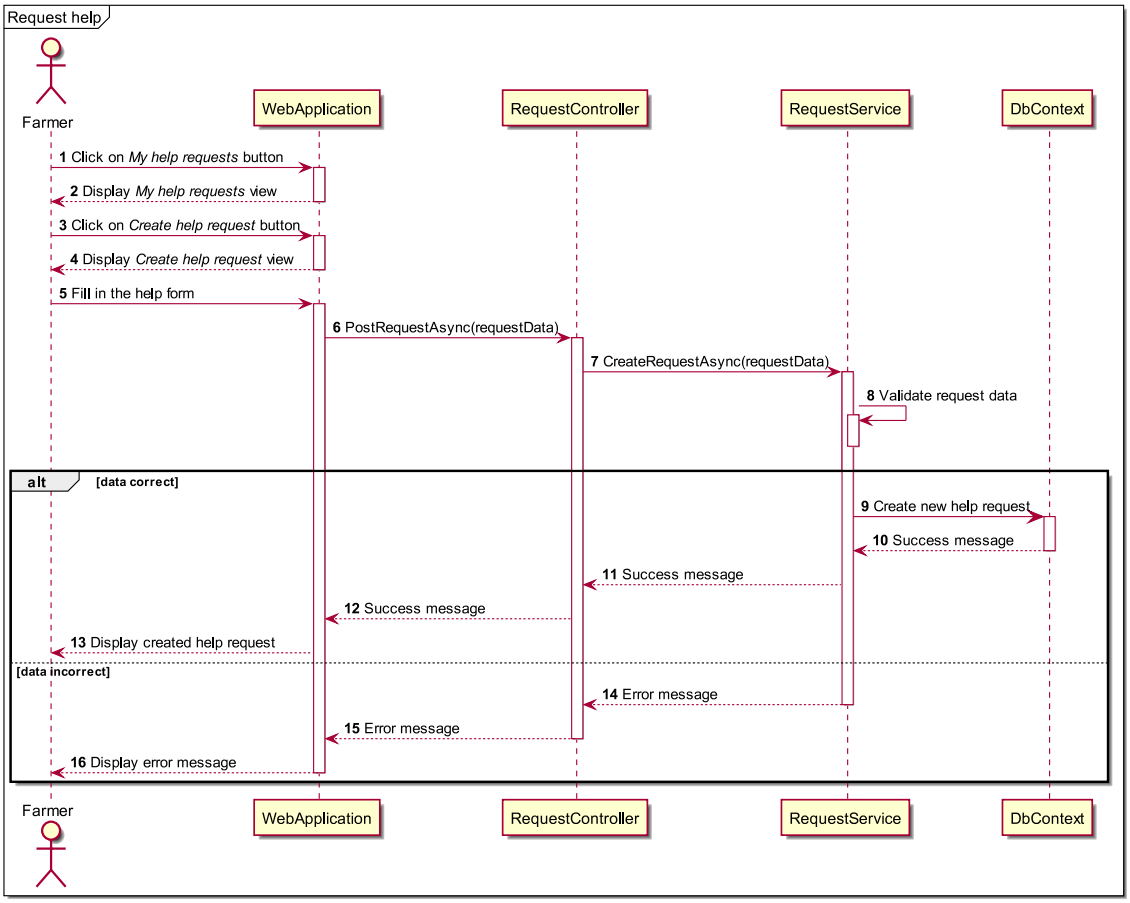


Figure 2.6: Sequence diagram presenting the creation of a help request.

### Set execution state of a daily plan

In the agricultural field, an agronomist acts as a liaison between farmers and policymakers. He provides advice on soil management and productivity to farmers. In the Telangana region, government agronomists visit farms in their areas of responsibility on a regular basis to gather information about farmers' performance. Thus, a possibility to organize all the appointments inside one interactive schedule is a vastly desired feature. It is realized via several components including *WebApplication*, *AgronomistController*, *AgronomistService*, *VisitsPlanner*, and *DbContext*. The interactions between them are depicted in the figure 2.8.

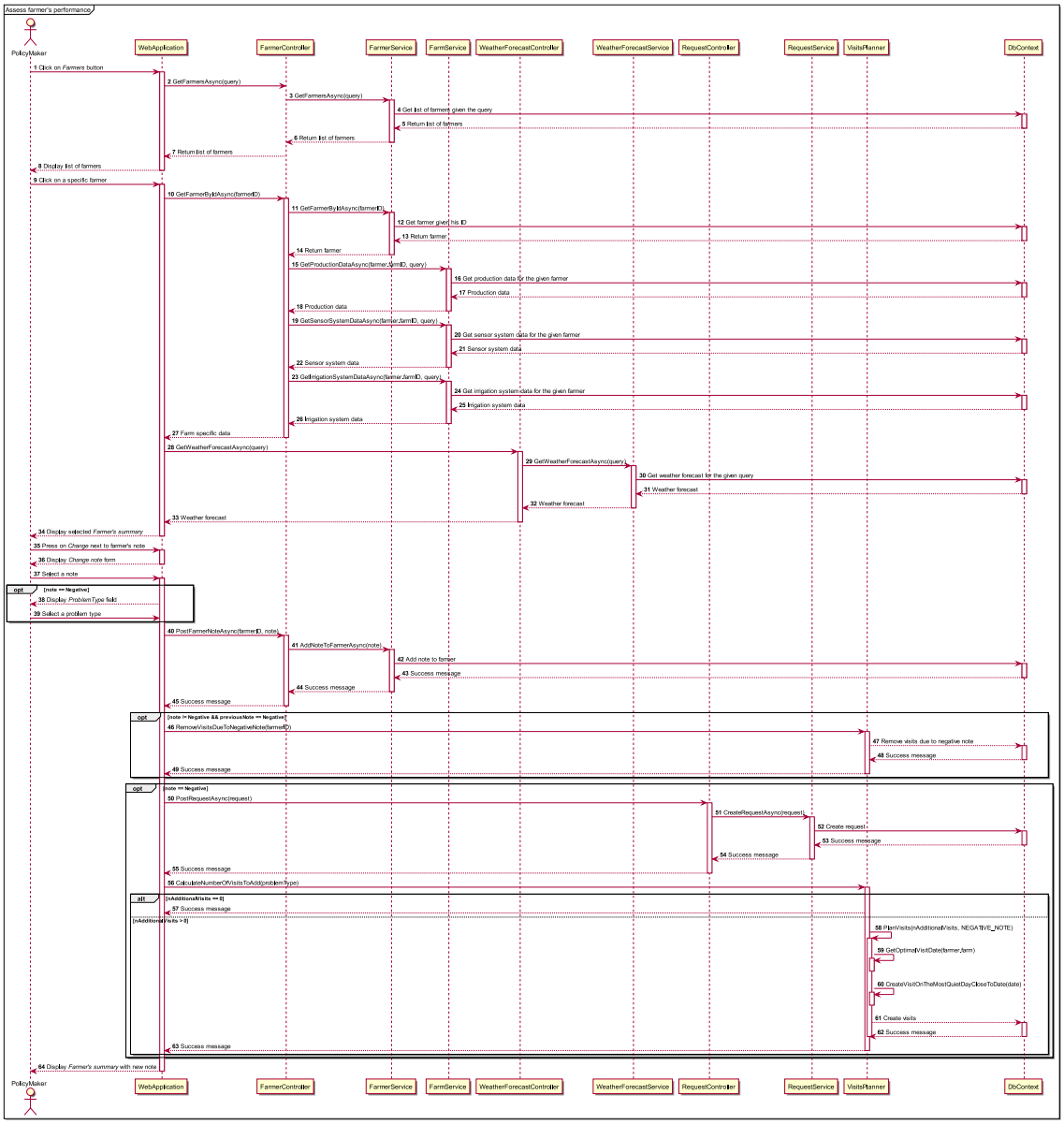


Figure 2.7: Sequence diagram presenting farmer's performance assessment.

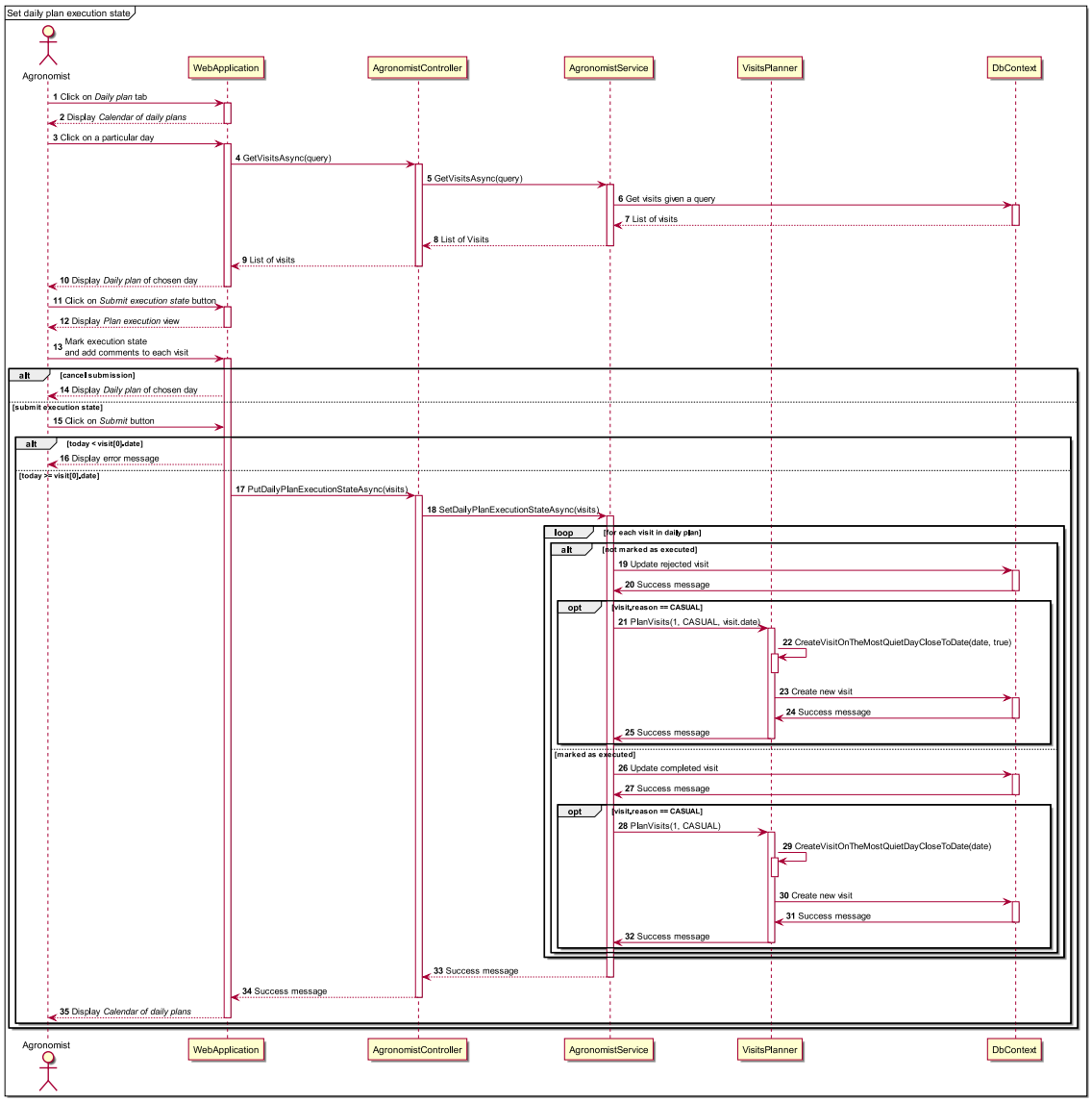


Figure 2.8: Sequence diagram presenting the action of displaying a daily plan.

## 2.5 Component Interfaces

The figures in this chapter depict methods implemented by the components of the DREAM server application presented previously in the figure 2.3.

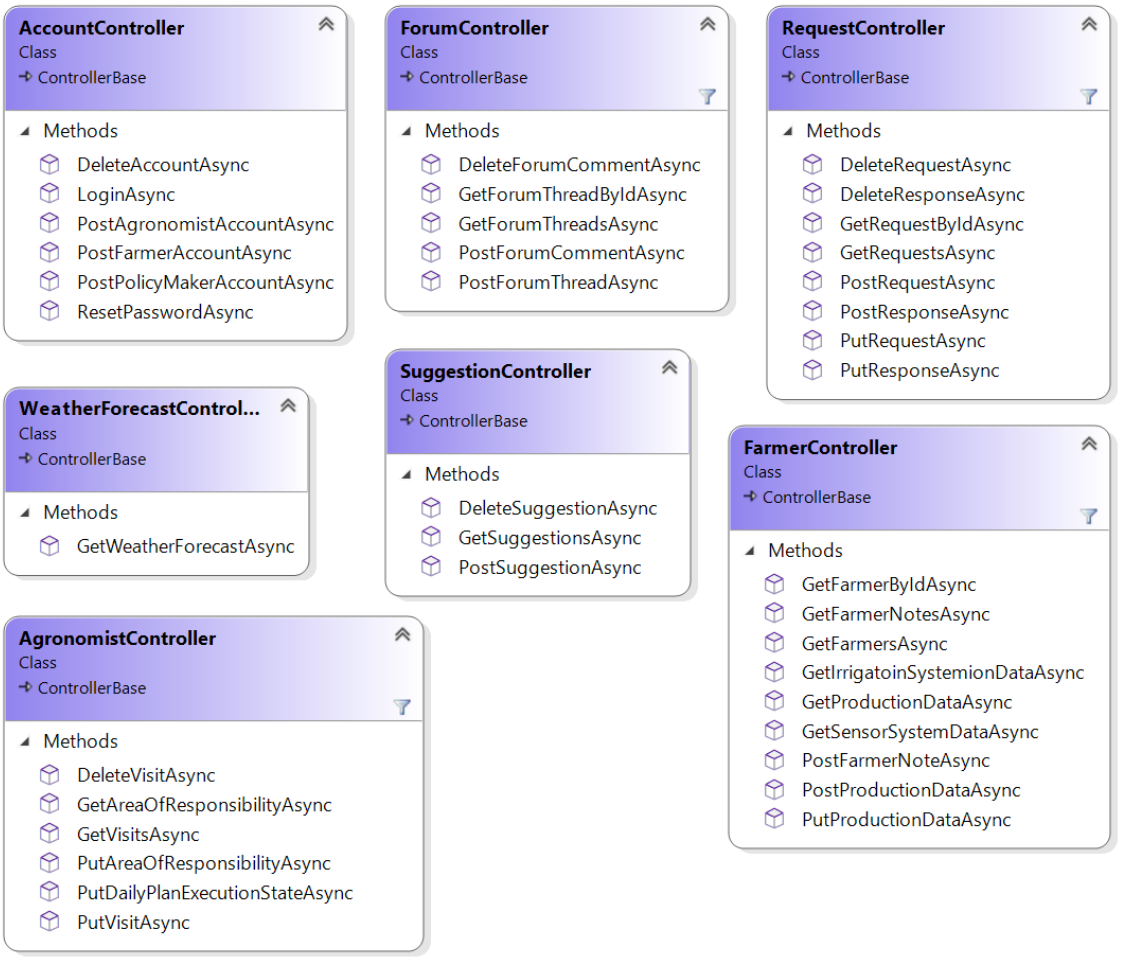


Figure 2.9: API components' interfaces

The first figure (2.9) represents controllers provided by the API layer to define HTTP endpoints exposed by the DREAM application server.

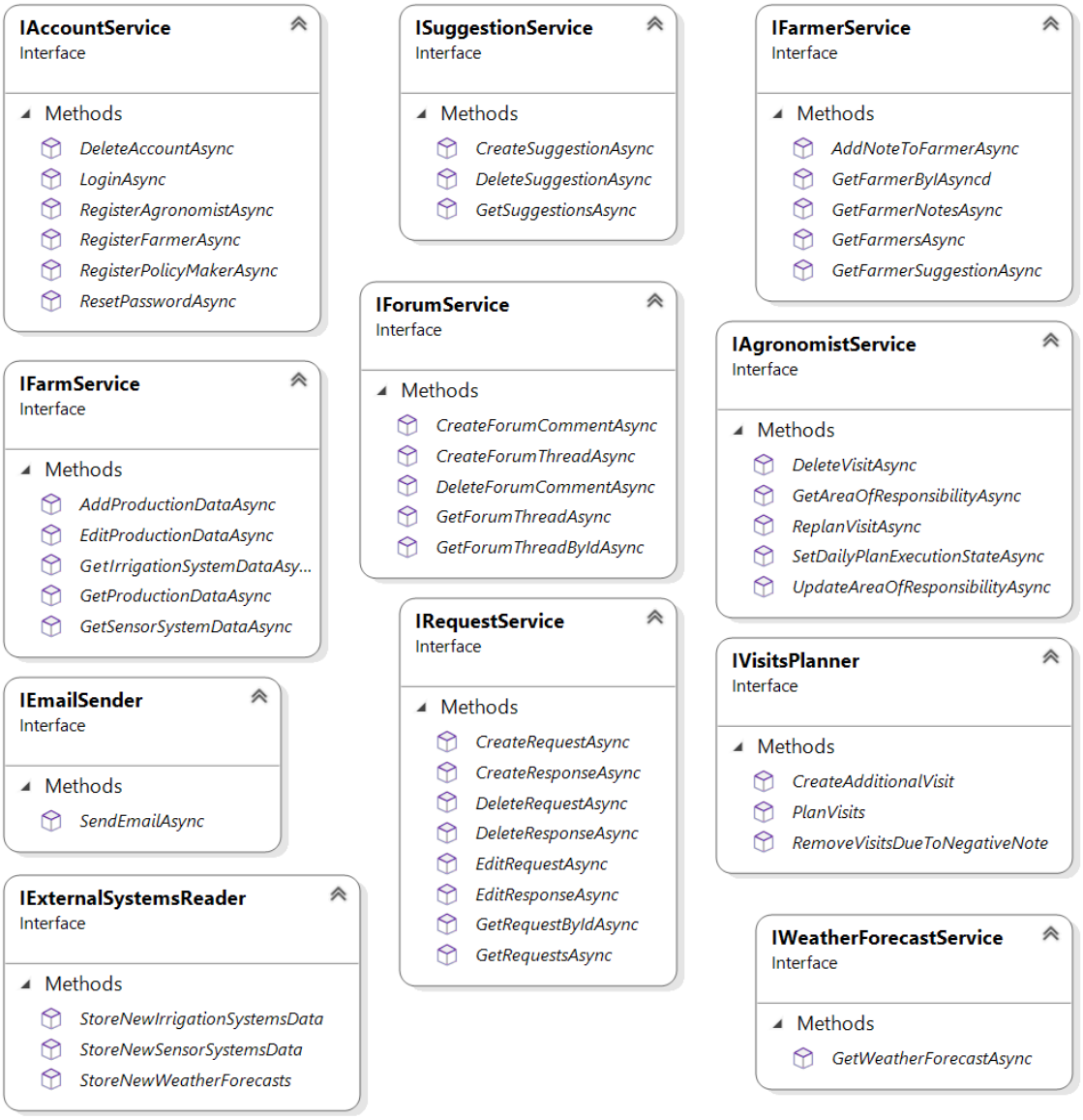


Figure 2.10: Business Logic components' interfaces

The second figure (2.10) presents interfaces that the components of the business logic layer implement. Specifically, the interfaces contain methods responsible for areas described in section (2.2.2) where DREAM server application components are presented. One of the implementation conventions that can be spotted here and in the previous figure is the use of *async* suffix for naming asynchronous methods in C#. It is an asynchronous design pattern recommended by Microsoft [7].

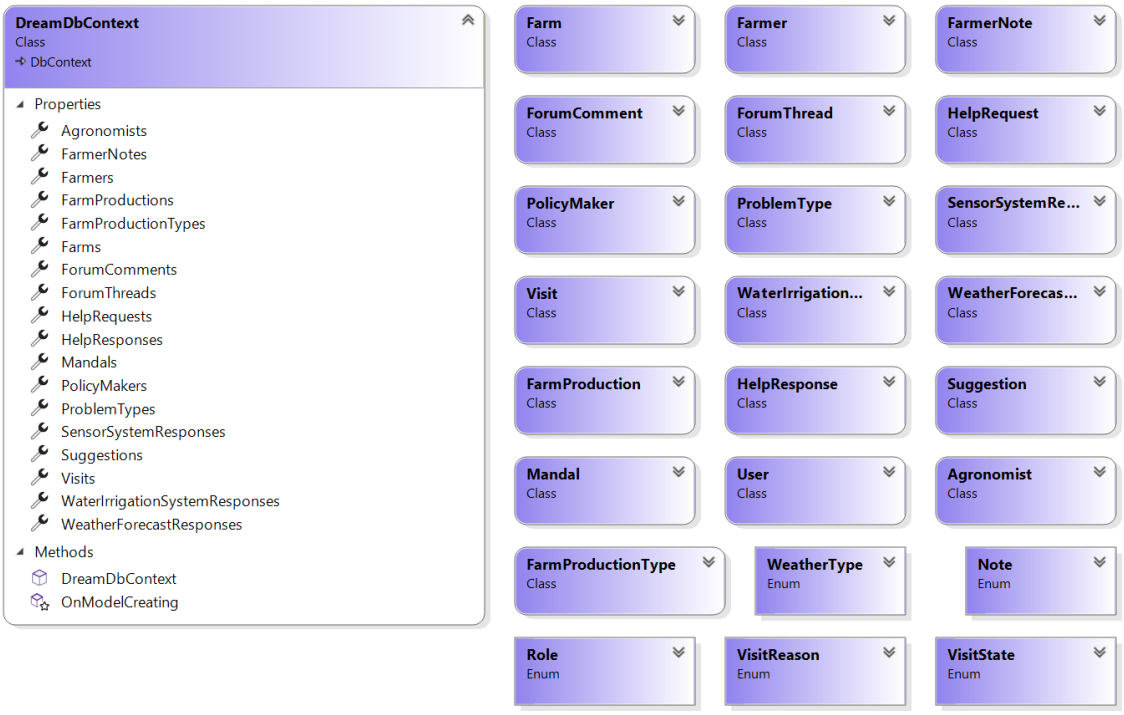


Figure 2.11: Data Access components' interfaces

The last figure 2.11 shows the classes of the Data Access layer. However, the details of classes presented in the figure are not shown to preserve readability. Those classes (apart from DreamDbContext class) do not contain any methods, since they serve for object-relational mapping [19]. Hence, each of them corresponds to at least one table in the database diagram (more than one in case of many-to-many relationships). Similarly, the fields of those classes correspond to the tables' columns. Detailed database diagram is presented in the following sections, in the figure 2.12. The only class that contains logic in this figure is the DreamDb-Context. It acts as a bridge between the database and the C# code – it contains fields that represent the tables of the database [9].

## 2.6 Selected Architectural Styles and Patterns

- **Client Server** - a pattern, which enforces the principle of separation of concerns, by separating the service requesters (clients) from the providers of a resource or service (servers). Clients and servers communicate over a network and can be hosted on separate hardware.
- **REST** (*Representational State Transfer*) – the API hosted on a DREAM server should be prepared using the REST API design rules [11] [6]. Firstly, it should not bind the client application to any specific implementation by utilizing HTTP as the communication protocol. Furthermore, as it was presented in the DREAM server components diagram in subsection 2.2.2, additional layers will be introduced to the server side application to avoid direct dependencies of the API on the database model. Additionally, the interface should be resource-oriented and the HTTP methods [18] should reflect the operations

done on the resources. All this aims at improving the maintainability, reusability, and readability of the server application.

- **N-tier application** – the server side application was divided into 3 layers: API, Business Logic, and Data Access. This architecture was adopted following the types of common web application architectures described in Microsoft's documentation [4]. However, the *User Interface* layer was replaced by *API* layer to better present its meaning. The choice was motivated by the simplicity of the architecture compared to other architectures like *Clean architecture* or microservices architecture.

## 2.7 Other Design Decisions

### 2.7.1 Visit Scheduling Algorithm

One of the most important features of the system is the ability to schedule agronomist's visits to a farm. Requirement **R5** (all the requirements are described in RASD) states that all farms must be visited at least twice a year, whereas **R14** extends it further by adding a constraint that under-performing farmer's, meaning farmers with a negative note, should be visited more often, depending on the problem they are facing.

The system must be therefore able to schedule visits to a farm in a way that satisfies both requirements. This section presents an overview of the algorithm employed to tackle this problem. All the listings provided include pseudocode that is intended to demonstrate only the high-level notion.

#### General approach

When planning a new visit it is necessary to take the following factors into consideration:

- dates and number of already planned visits to the farm,
- number of visits planned for an agronomist on a given day,
- type of the problem the farmer is facing.

Function *calculateVisitsToAdd* presented in the listing 2.1 calculates the number of visits to add to the farm, given the number of already planned visits to the farm, the number of visits planned for an agronomist on a given day and the type of the problem the farmer is facing. In case there are already more visits to the farm planned than the number of casual visits (2) and the number of visits to add, then no new visit is necessary and the function returns 0. Such situation may happen when there are many visits planned to one farm due to the agronomist's decision.

Listing 2.1: Function calculating the number of visits to add due to a problem.

```
1 static const int DEFAULT_NUM_OF_CASUAL_VISITS = 2;
2
3 int calculateVisitsToAdd(ProblemType problemType) {
4     int agr_dec_visits = 0;
```

```
5     vector<Visit> visits = getFutureVisitsOnFarm(farmer.farm);
6
7     if (visits.length >= DEFAULT_NUM_OF_CASUAL_VISITS + problemType.nAdditionalVisits)
          ↪ {
8         return 0;
9     }
10
11    for (const auto &visit : visits) {
12        if (visit.reason == VisitReason.AGRONOMIST_DECISION) {
13            agr_dec_visits++;
14        }
15    }
16    int nVisitsToAdd = problemType.nAdditionalVisits - agr_dec_visits;
17
18    return nVisitsToAdd;
19 }
```

Another function, presented in the listing 2.2 is *getOptimalVisitDate*. It finds the greatest gap between two consecutive visits to the farm and inserts a new visit in between. It also considers a time slot in exactly one year from the current date, so that in case of completing a casual visit, a new one is created in approximately half of a year.

Listing 2.2: Function responsible for picking the optimal date of a new visit.

```
1  DateTime getOptimalVisitDate(Farm &farm) {
2      // Make sure that the visits are in ascending ordered
3      vector<Visit> visits = getFutureVisitsOnFarm(farm);
4      DateTime greatestGap = 0;
5      DateTime optimalVisitDate;
6
7      for (int i = 0; i < visits.length - 1; i++) {
8          if ((visits[i + 1].date - visits[i].date) > greatestGap) {
9              greatestGap = visits[i + 1].date - visits[i].date;
10             optimalVisitDate = visits[i].date + greatestGap / 2;
11         }
12     }
13
14     if ((DateTime.now() + DateTime(year=1) - visits[visits.length - 1].date) >
           ↪ greatestGap) {
15         greatestGap = DateTime.now() + DateTime(year=1) - visits[visits.length - 1].
               ↪ date;
16         optimalVisitDate = visits[visits.length - 1].date + greatestGap / 2;
17     }
18
19     return optimalVisitDate;
20 }
```

### Planning additional visits

This section presents the process of scheduling a new visit to a farm. Function *createVisitOnTheMostQuietDayCloseToDate* is presented in the listing 2.3. It is responsible for creating a new visit to the farm on the most quiet day close to the date specified in the function's argument. It goes through up to 5 days before the proposed date and searches for an agronomist, who has not reached the maximum number of visits to the farm on a given day, otherwise, the visit is scheduled for the agronomist with the least visits planned in the specified window.

Listing 2.3: Function creating a new visit on the most quiet day that is close to the specified date.

```
static const int MAX_VISIT_DATE_SHIFT = 30; // days
static const int MAX_REPLANNED_CASUAL_VISIT_DATE_SHIFT = 5; // days
static const int MAX_DAILY_VISITS_PER_AGRONOMIST = 5; //days

Visit createVisitOnTheMostQuietDayCloseToDate(DateTime dateTime, bool
    ↪ isCasualVisitRejected = False) {
    Visit newVisit;
    int leastNumOfVisits = agronomists[0].getVisits(dateTime).length;
    newVisit.agronomist = agronomists[0];
    newVisit.date = dateTime;

    if (leastNumOfVisits <= MAX_DAILY_VISITS_PER_AGRONOMIST) {
        return newVisit;
    }

    int dateShift = isCasualVisitRejected ? MAX_REPLANNED_CASUAL_VISIT_DATE_SHIFT
                  : MAX_VISIT_DATE_SHIFT;

    for (const auto &agronomist : agronomists) {
        for (int i = 1; i < dateShift; i++) {
            DateTime proposedDateTime = dateTime - DateTime(days=i);
            vector<Visit> visits = agronomist.getVisits(proposedDateTime);

            if (visits.length < leastNumOfVisits) {
                leastNumOfVisits = visits.length;
                newVisit.agronomist = agronomist;
                newVisit.date = proposedDateTime;
            }

            if (leastNumOfVisits <= MAX_DAILY_VISITS_PER_AGRONOMIST) {
                return newVisit;
            }
        }
    }

    return newVisit;
}
```

Function *createAdditionalVisits*, presented in the listing 2.4, is the main function responsible for scheduling additional visits to the farm.

Listing 2.4: Function creating additional visits.

```
Visit createAdditionalVisit() {
    Visit newVisit;
    DateTime optimalVisitDate = getOptimalVisitDate(farmer.farm);
```

```
4     newVisit = createVisitOnTheMostQuietDayCloseToDate(optimalVisitDate);
5
6     return newVisit;
7 }
```

Due to the abovementioned requirements, the system must be able to schedule different types of visits to the farm. These types are identified by the visit's reason. The following listing 2.5 presents the function meant to handle this problem.

Listing 2.5: Function responsible for planning new visits.

```
1 void Farmer::planVisits(int n_visits, VisitReason reason, previousDate = nullptr) {
2     if (reason == VisitReason.CASUAL) {
3         for (int i = 0; i < n_visits; i++) {
4             DateTime dateTime;
5             Visit newVisit;
6
7             if (previousDate != nullptr) {
8                 dateTime = previousDate + DateTime(days=
                      ↪ MAX_REPLANNED_CASUAL_VISIT_DATE_SHIFT);
9                 newVisit = createVisitOnTheMostQuietDayCloseToDate(dateTime, True);
10            } else {
11                DateTime mostFuture = getDateTimeOfMostFutureVisit(VisitReason.CASUAL);
12                dateTime = mostFuture + DateTime(months=6);
13                newVisit = createVisitOnTheMostQuietDayCloseToDate(dateTime);
14            }
15
16            newVisit.reason = reason;
17            VisitDataAccess.createVisit(newVisit);
18        }
19    } else if (reason == VisitReason.NEGATIVE_NOTE) {
20        for (int i = 0; i < n_visits; i++) {
21            Visit visit = createAdditionalVisit();
22            visit.reason = reason;
23            VisitDataAccess.createVisit(visit);
24        }
25    }
26 }
```

### Farmer registration

Listing 2.6 presents the process of creating a farmer. Each time a new farmer is created, the system must plan two casual visits to his farm.

Listing 2.6: Function responsible for creating a new farmer.

```
1 Farmer createFarmer(string name) {
2     Farmer farmer(name);
3     farmer.planVisits(DEFAULT_NUM_OF_CASUAL_VISITS, VisitReason.CASUAL);
4
5     return farmer;
6 }
```

### Daily plan submission

Function *updateVisit* (listing 2.7) is responsible for changing the date of a visit. It is only possible to do so only if the new date is greater or equal than the current one. In case an agronomist wants to reschedule a visit that is casual, the system allows him to do so, but the date can be postponed only by a maximum of 5 days. Otherwise, the systems rejects the request and throws a warning.

Listing 2.7: Function responsible for updating a visit.

```
1  void updateVisit(Visit &originalVisit, Visit &newVisit) {
2      if (newVisit.date >= DateTime.now()) {
3          return;
4      }
5
6      if (newVisit.reason == VisitReason.CASUAL) {
7          if ((newVisit.date - originalVisit.date) >
               ↪ MAX_REPLANNED_CASUAL_VISIT_DATE_SHIFT) {
8              throw Warning;
9          }
10         newVisit.farm.farmer.planVisits(1, newVisit.reason);
11     }
12
13     newVisit.farm.farmer.planVisits(1, newVisit.reason);
14 }
```

The process of submitting a visit is described in the listing 2.8. In case the date of a visit is greater than the current date, the visit is submitted. In case the date is less than the current date, a warning is thrown. A casual visit is a cyclic one, therefore a new one is scheduled every time its status becomes confirmed or rejected. In case of rejecting a casual visit, a new one is scheduled in the maximum of next 5 days following the process shown in the listing 2.5.

Listing 2.8: Function responsible for submitting a visit.

```
1  void submitVisit(Visit &visit) {
2      if (visit.date < DateTime.now()) {
3          return;
4      }
5
6      if (visit.state == VisitState.CONFIRMED) {
7          if (visit.reason == VisitReason.CASUAL) {
8              VisitDataAccess.updateVisitData(visit); // update record inside the
                   ↪ database
9              visit.farm.farmer.planVisits(1, VisitReason.CASUAL);
10         }
11     } else if (visit.state == VisitState.REJECTED) {
12         VisitDataAccess.updateVisitData(visit);
13
14         if (visit.reason == VisitReason.CASUAL) {
15             visit.farm.farmer.planVisits(1, visit.reason, visit.date);
16         } else {
17             visit.farm.farmer.planVisits(1, visit.reason);
18         }
19     }
20 }
```

#### Obtaining a negative note

Every time a farmer's note changes from negative to neutral or positive, the system deletes all the visits to the farm that were planned due to the negative note. The function *DeleteVisitsCreatedDueToNegativeNote* (listing 2.9) handles that.

Listing 2.9: Function that deletes visits created due to a negative note.

```
1 void deleteVisitsCreatedDueToNegativeNote() {
2     visits.erase(remove_if(
3         visits.begin(), visits.end(),
4         [](const Visit &visit) {
5             return visit.reason == VisitReason.NEGATIVE_NOTE;
6         }), visits.end());
7 }
```

Function presented in the listing 2.10 assures that every time a farmer receives a negative note, there are additional visits to his farm planned. The number of these visits depends on the type of the problem the farmer faces, which is priorly specified by a policy maker when assigning the note.

Listing 2.10: Function responsible for setting the farmer's note.

```
1 void Farmer::setNote(Note note, ProblemType problemType = nullptr) {
2     if (this.note != Note.NEGATIVE && note == Note.NEGATIVE) {
3         this.note = note;
4
5         int nVisitsToAdd = calculateVisitsToAdd(problemType);
6         planVisits(nVisitsToAdd, VisitReason.NEGATIVE_NOTE);
7     } else {
8         if (this.note == Note.NEGATIVE) {
9             deleteVisitsCreatedDueToNegativeNote();
10        }
11        this.note = note;
12    }
13 }
```

### 2.7.2 Database model

This section presents a database model devised for the DREAM application. One of the design decisions included in the figure (2.12) is to use one table per type configuration [8]. The configuration is used when modelling the class *User*, which is the base of *PolicyMaker*, *Agronomist* and *Farmer* classes. The decision was motivated by the great number of relationships that each of those derived classes has. With three separate classes for each role, these relationships will be easier to manage and maintain since creation of one, huge *User* class is avoided. The major drawback of this decision is the need to perform additional queries to determine the role of a user during the log in process. To tackle this issue, a column *Role* was introduced to the *User* table. However, such column may introduce consistency issues because the *Role* saved in a given row of the User's table must correspond to a foreign key *UserId* in an appropriate table (*PolicyMaker*, *Agronomist* or *Farmer*). This consistency issue is addressed by the implementation of the DREAM server application.

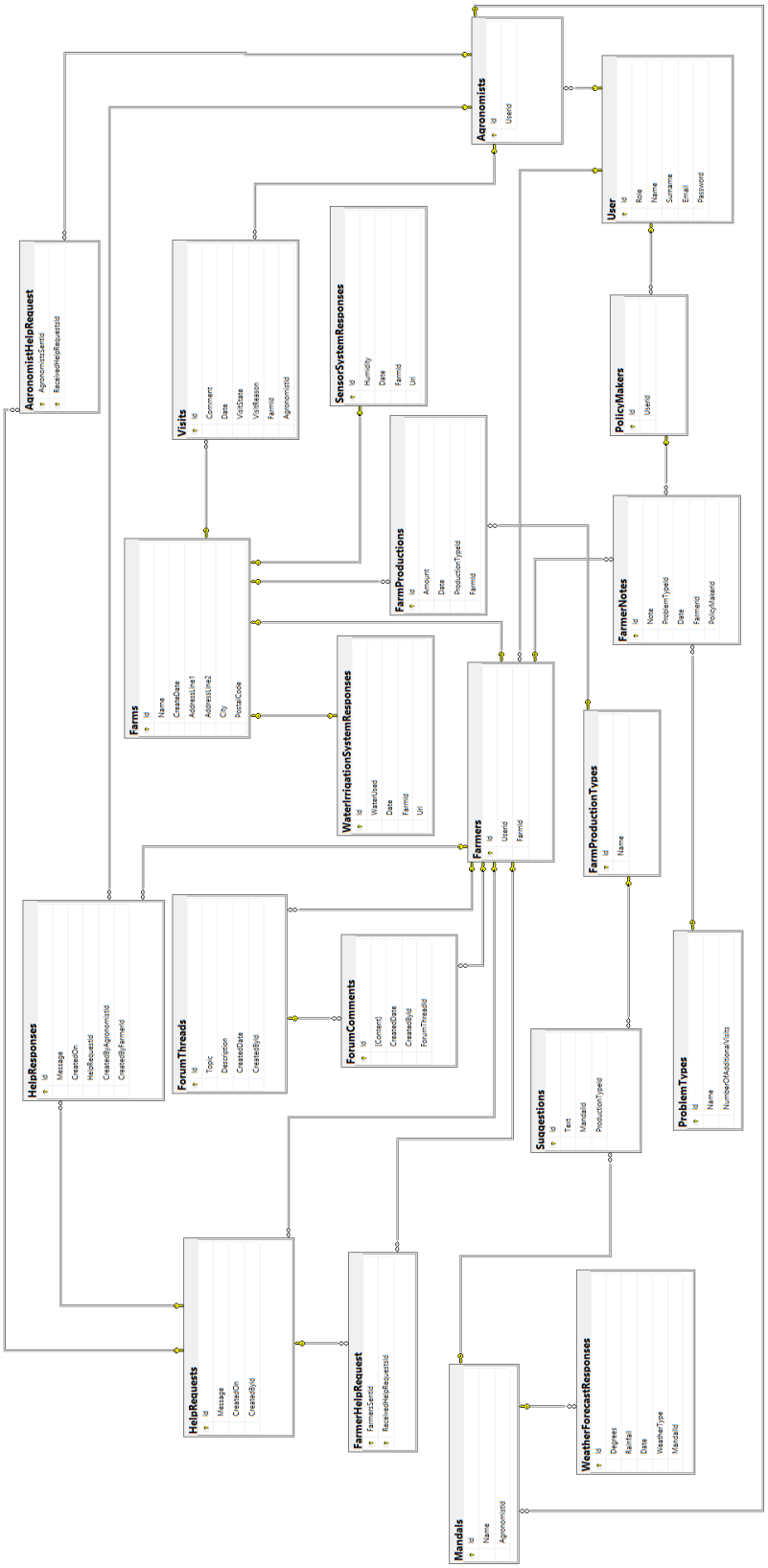


Figure 2.12: Database model

### 2.7.3 Authentication and authorization method

The DREAM system will use JSON Web Tokens (JWT) [3] for authentication and authorization. JWT is a standard that defines a way for transmitting information. Each token is digitally signed, allowing the consumer to verify if the token is correct and has not been forged. JWTs are stateless by design, which works well with the statelessness of HTTP protocol.

Using JSON Web Tokens affects the system in several ways. They come in two types: the access token and the refresh token. The refresh token is obtained using an API call, containing the user's e-mail and password. Each of the tokens has a lifetime – when the access token is no longer valid, we use the refresh token to obtain a new one. When the refresh token loses its validity, the user is automatically signed out.

The tokens have to be provided in a header of every API call, apart from sign-up, sign-on and remind password requests. They are stored in a local storage of user's browser. JWT can not be invalidated, so sign-out results in clearing the local storage and redirecting the user back to the home page.

### 2.7.4 WebApplication static content generation

The WebApplicaion tier consist of a Nginx server, hosting the production build of the website [10]. The production build has to be generated from the code using a separate NodeJS environment during build phase and then passed to the Nginx process using a shared volume.

The application can be also hosted directly using NodeJS, although it is not suited for production use and has limited scalability.

### 2.7.5 Dependency Injection

Dependency injection, also known as dependency inversion technique, focuses on providing implementations of required interfaces instead of creating them explicitly by the dependent objects [5]. The implementation of DREAM Web Application will leverage it to address the difficulty of testing the business logic layer [4].

# Chapter 3

# User Interface Design

DREAM is a web-based application. It could be utilized on a variety of devices, independent of form factor. The application, however, is expected to be largely utilized on desktop computers. As a result, the given mockups depict representations of the application's desktop version.

This chapter provides pictures of all the mockups, that will be later used for the requirements traceability (see chapter 4).

### Unregistered user

The user can navigate to the sign-up screen using *Create account* option, available in the horizontal navigation bar on the top of the page. After correctly filling the form, the user is redirected to the login page and receives a notification informing him about the success of the account creation process.

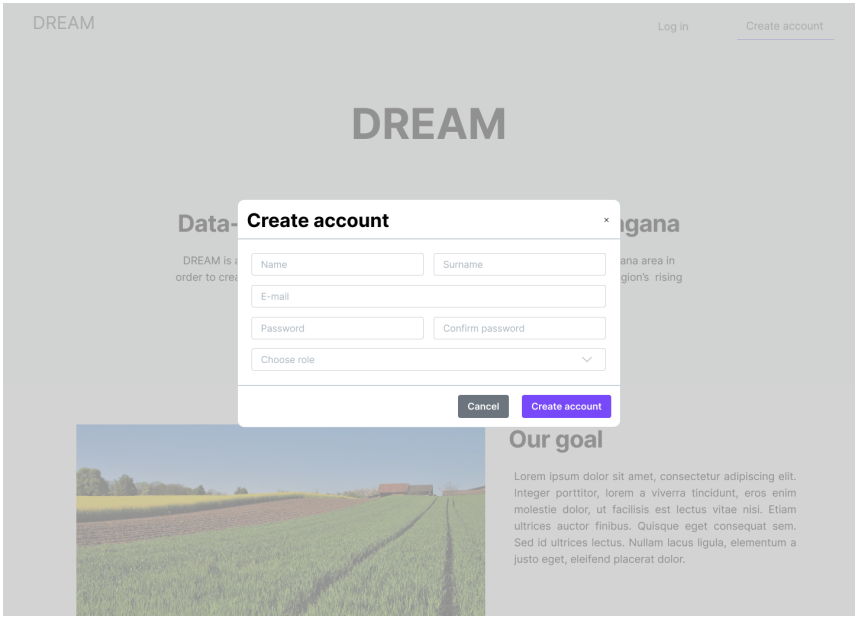


Figure 3.1: **M1.** User registration.

When the user chooses *Farmer* role, additional fields appear, namely Sensor system's ID, Water irrigation system's ID, Address line 1, Address line 2, Postal code, City, and Mandal.

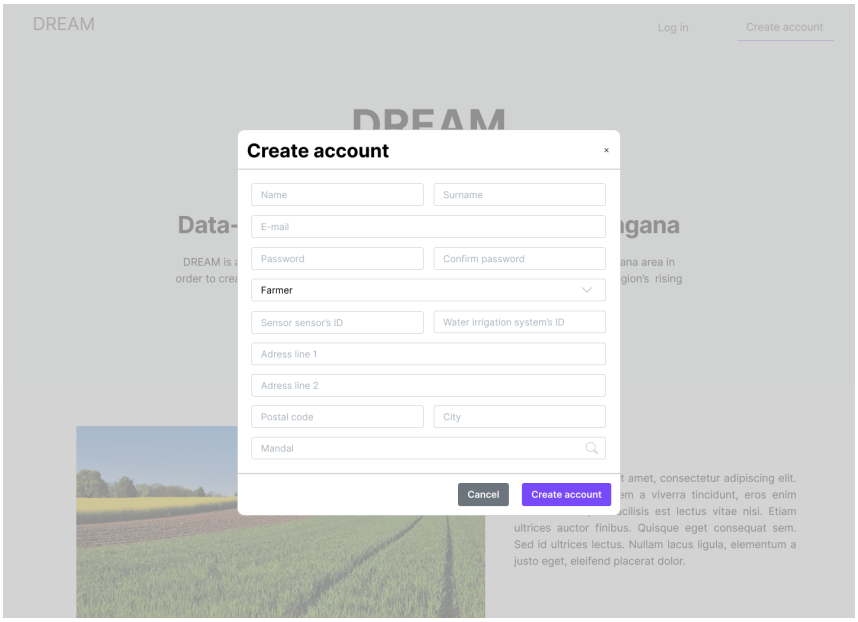


Figure 3.2: **M2.** Farmer registration.

When the user chooses *Agronomist* role, additional fields appear, allowing him to choose the area of responsibility.

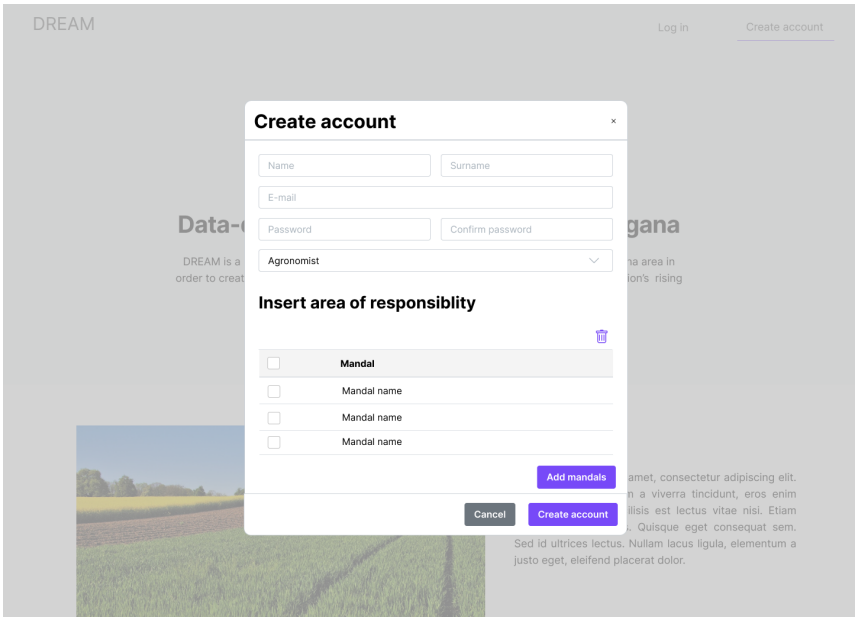


Figure 3.3: **M3.** Agronomist registration.

The user can navigate to the log in screen using *Log in* option, available in the horizontal navigation bar on the top of the page. It contains fields for providing e-mail and password.

The user can also choose an option to remind a password, via a button in the lower part of the modal dialog.

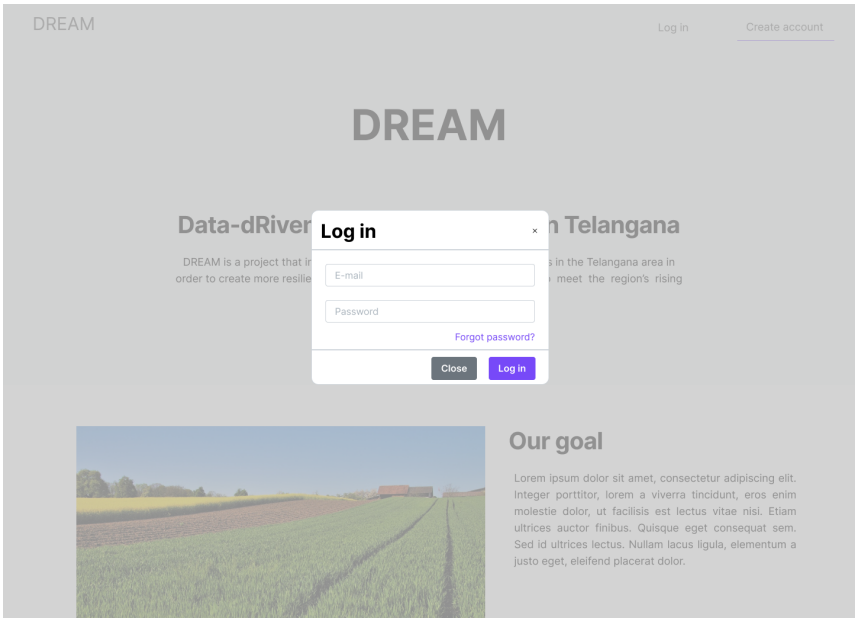


Figure 3.4: **M4.** Log in view.

The user can reset his password using the *Remind password* view. He has to provide a valid e-mail address, to which the message will be sent.

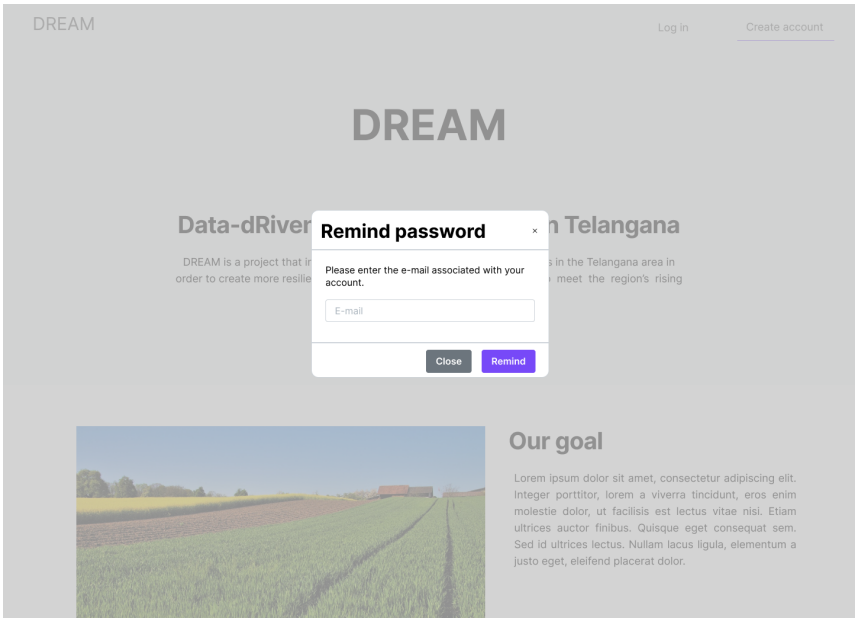


Figure 3.5: **M5.** Remind password.

### Farmer

Regardless of current view, a farmer can navigate to his summary, by clicking on the button containing his name and surname in the top navigation bar. The summary contains all the information about his account and farm. Following information are provided: account data, note history, production data, farm visits, weather, soil humidity, water usage, and help requests. User can see detailed view about farm visits and help requests, by clicking on the eye button in a given table row. Farmer can also delete his account, using the *Delete account* button in the bottom of the page and after confirming his choice in a modal pop-up.

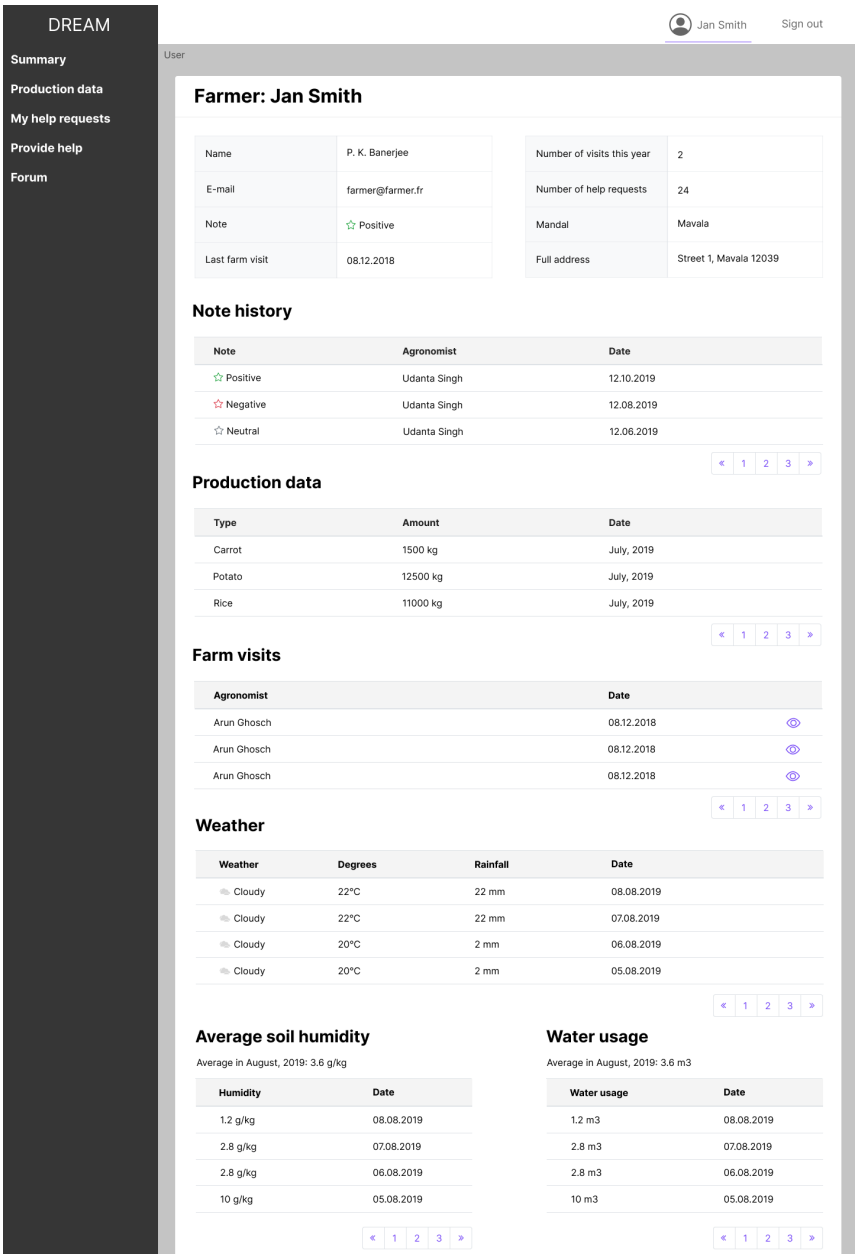


Figure 3.6: **M6.1.** Farmer's user view - part 1.

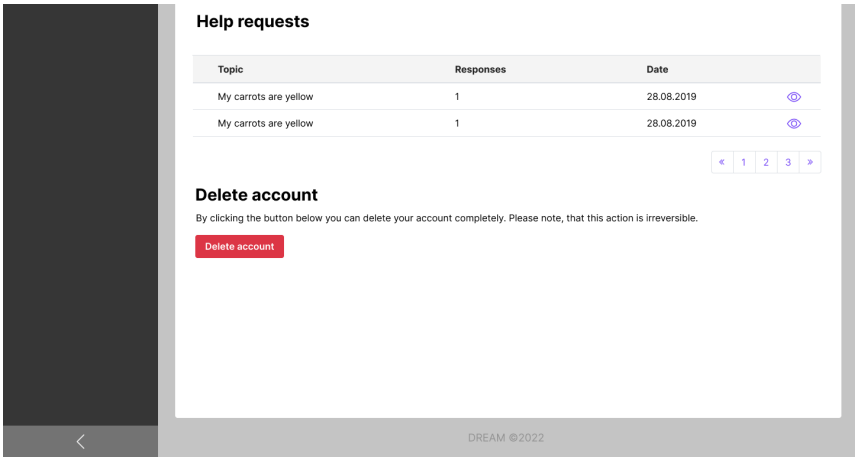


Figure 3.7: **M6.2.** Farmer's user view - part 2.

Directly after logging in, farmer is forwarded to the dashboard page, containing a summary of the most important information. From there he can navigate to other views by the navigation bar in the left part of the page, the navigation bar in the upper part of the page or by clicking on the button in the lower part of some summary cards.

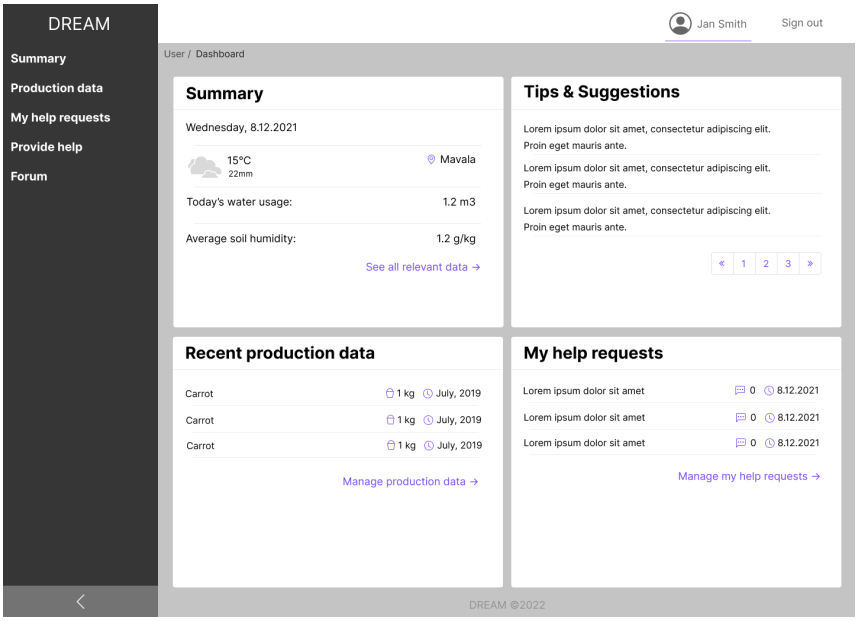


Figure 3.8: **M7.** Farmer's dashboard.

The farmer can manage his production data, by choosing the option *Production data* in the left navigation bar. The production data view consists of a table with the latest data, an *Add new* button and an option to delete previously provided information. After clicking on the *Add new* button, a modal dialog appears, allowing the farmer to create a new entry.

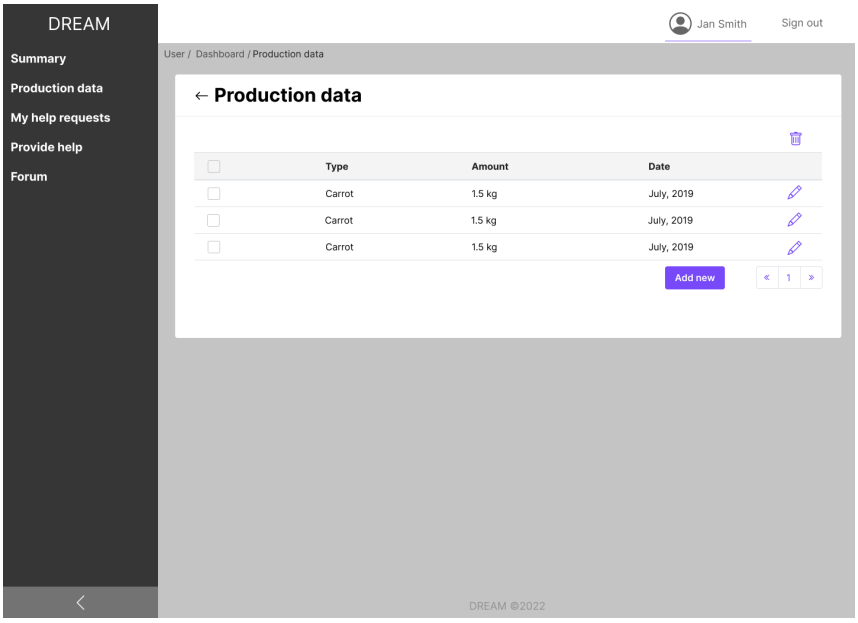


Figure 3.9: **M8.** Farmer's production data.

The farmer can see all his help requests, by clicking on *My help requests* option in the left navigation bar. Help requests are visualized in the form of a table, with entries containing a short summary of given help requests. User can view detailed information by clicking on given entry or create a new help requests, by clicking on *Create help request* button. The farmer can also search for a help requests by topic, using a search bar in the upper part of the content card.

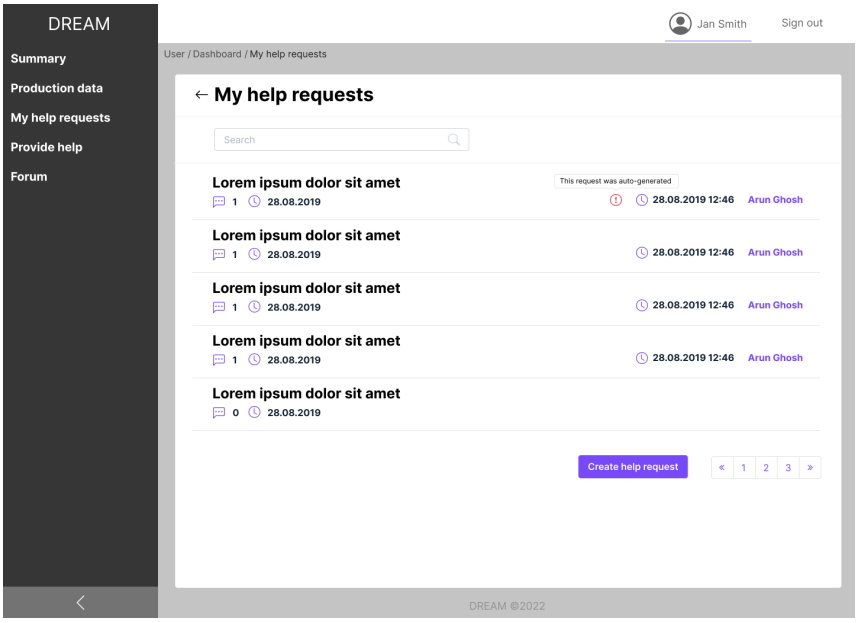


Figure 3.10: **M9.** Help requests created by farmer.

After clicking on a given entry in the help request table, the user is redirected to a view,

containing all detailed information about the help requests, as well as advice provided by other farmers and agronomists.

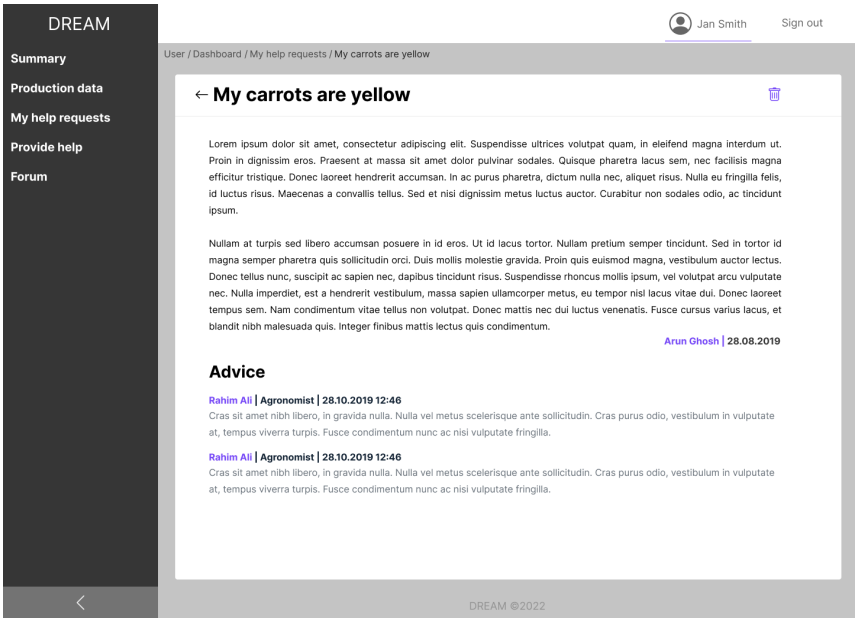


Figure 3.11: **M10.** Specific help request created by farmer.

After clicking on the *Create help request* button in the help requests view, a modal dialog appears, allowing the farmer to create a new help request, by filling in the topic and message.

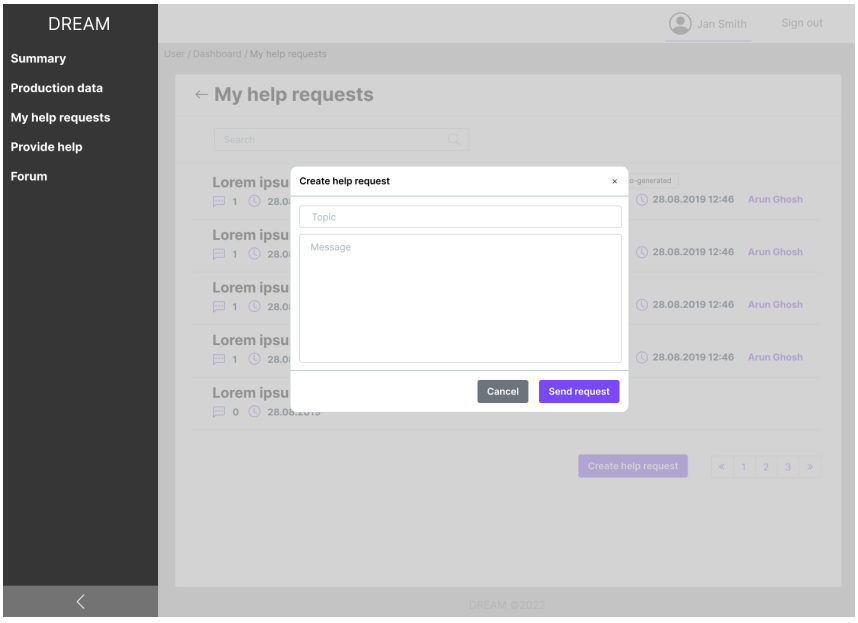


Figure 3.12: **M11.** Creating help request.

The farmer with positive note can see all received help requests, by clicking on *Provide help* option in the left navigation bar. Help requests are visualized in the form of a table, with entries

containing a short summary of given help requests. User can view detailed information by clicking on a given entry. The farmer can search for a help requests by topic, using a search bar in the upper part of the content card.

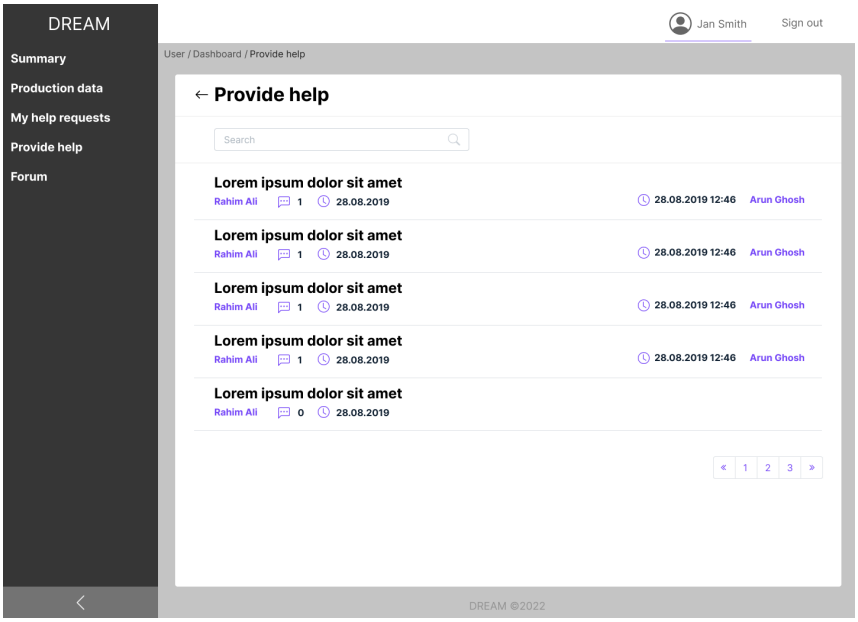


Figure 3.13: **M12.** Help requests received by farmer with positive note.

After clicking on a given entry in the help request table, the user is redirected to a view, containing all detailed information about the help requests and allowing the farmer to create a new advice, by filling a short form on the bottom of the page. This view is also accessible for agronomists, by clicking on the *Provide help* button in agronomist's left navigation bar.

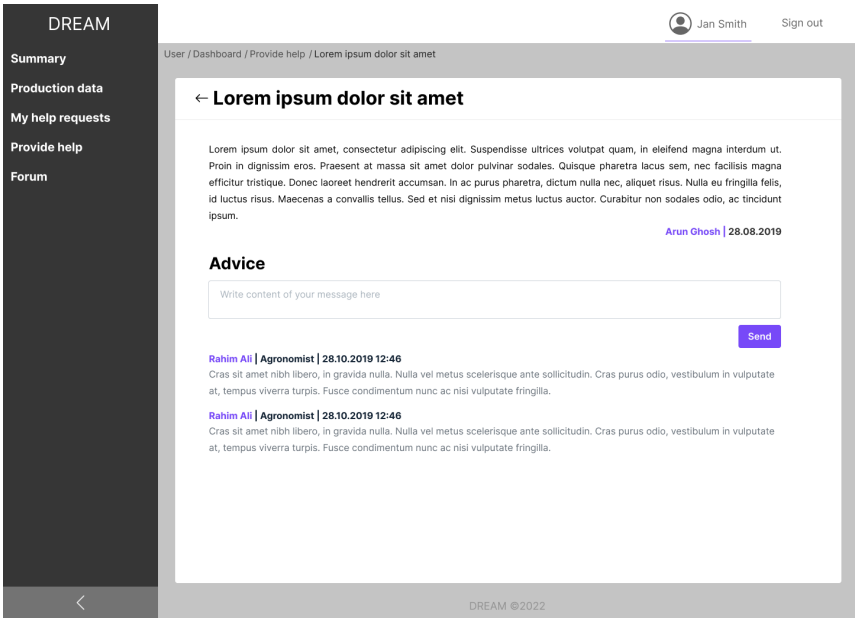


Figure 3.14: **M13.** Specific help request received by farmer with positive note (applicable also for the agronomist).

The farmer can access the forum using *Forum* button in the left navigation bar. The forum is built similarly to the help requests view – it consists of a table, with entries showing a short summary of a given thread. The user can search for a thread by name using the search bar, located in the upper part of the content card, display a detailed thread view, by clicking on the entry or create a new forum thread, using the *Create forum thread* button in the lower part of the content card.

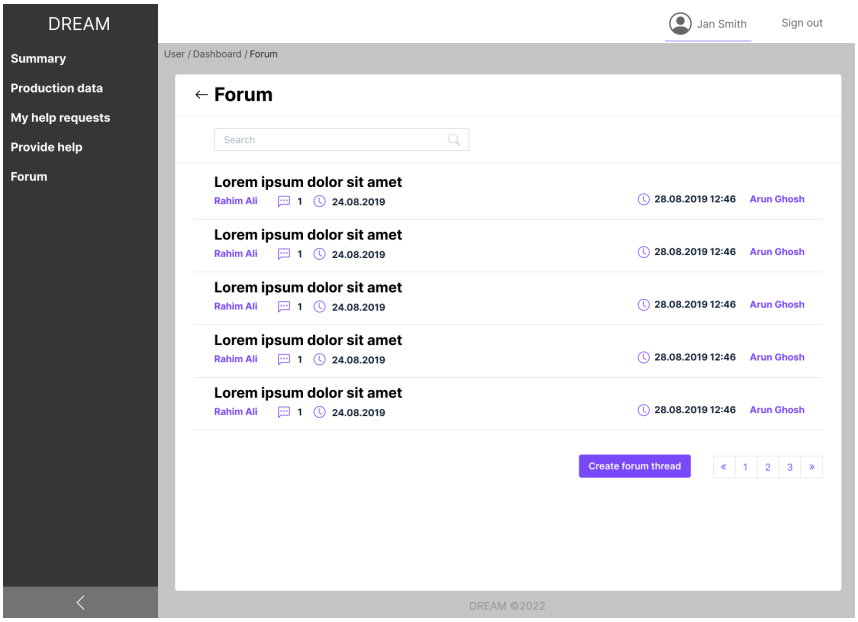


Figure 3.15: **M14.** Forum view.

After clicking on a given entry in the forum thread table, the farmer is redirected to a detailed thread view. The user can create a comment, by filling a short form in the lower part of the content card or delete a previously created comment, using the thrash icon next to the entry.

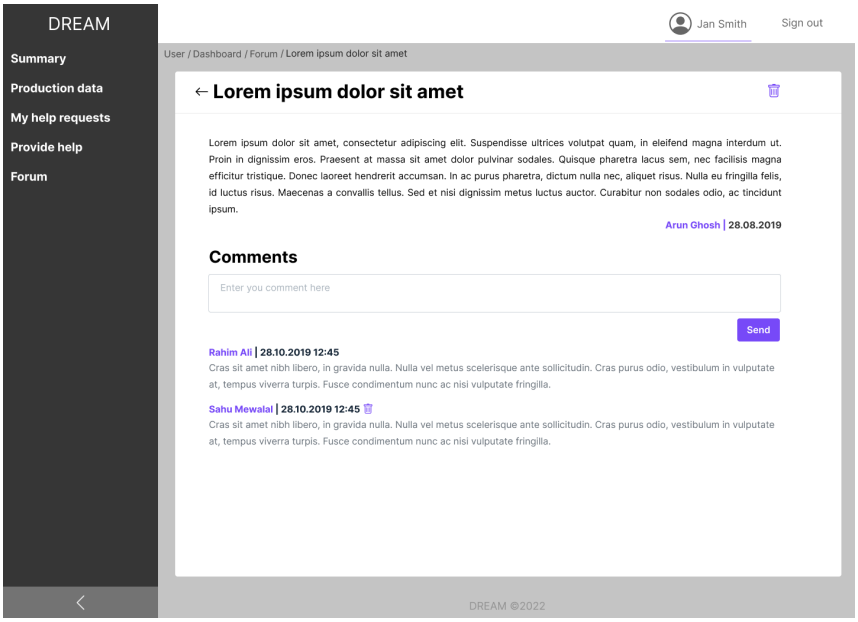


Figure 3.16: **M15.** Specific forum thread.

After clicking on the *Create forum thread* button in the help requests view, a modal dialog appears, allowing the farmer to create a new forum thread, by filling topic and message.

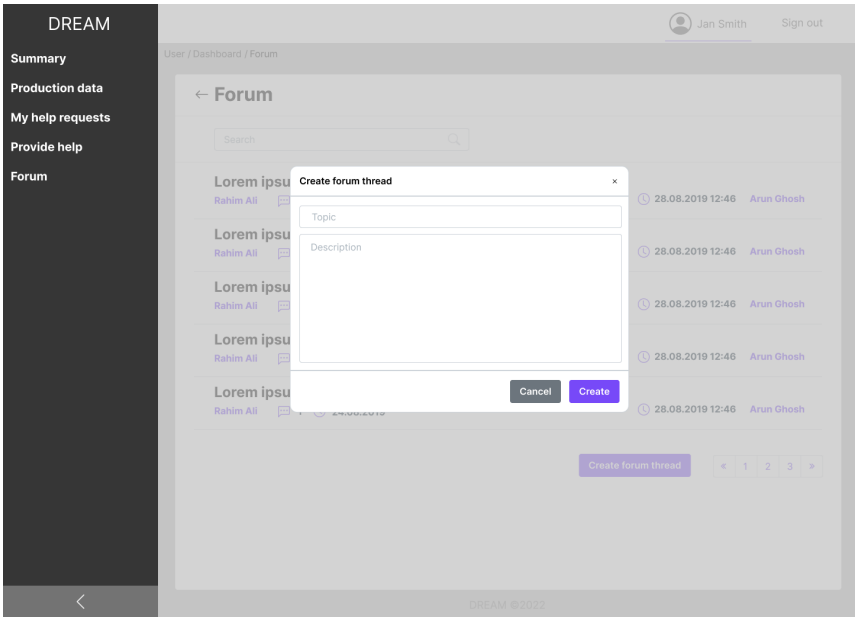


Figure 3.17: **M16.** Creating forum thread.

### Agronomist

The agronomist can open his account summary, by clicking on the button in the top navigation bar, containing his name and surname. In the user view, he can view his account details, manage his area of responsibility and delete his account.

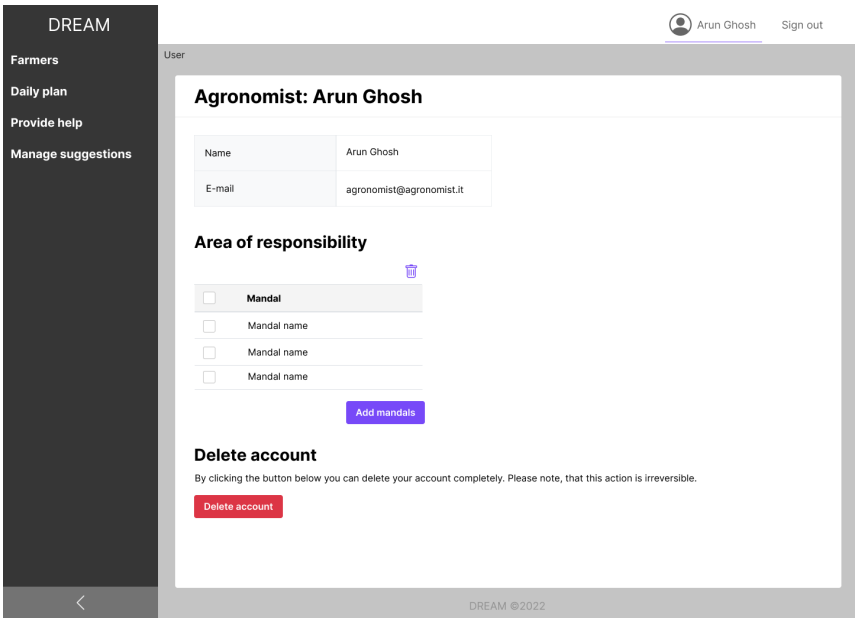


Figure 3.18: **M17.** Agronomist's user view.

Directly after logging in, an agronomist is redirected to the dashboard view, which contains a summary of the most important information, such as a short list of farmer in his area of responsibility, today's visits and last help requests received. He can navigate to other views using the left and top navigation bar or detailed button in the lower part of the content cards.

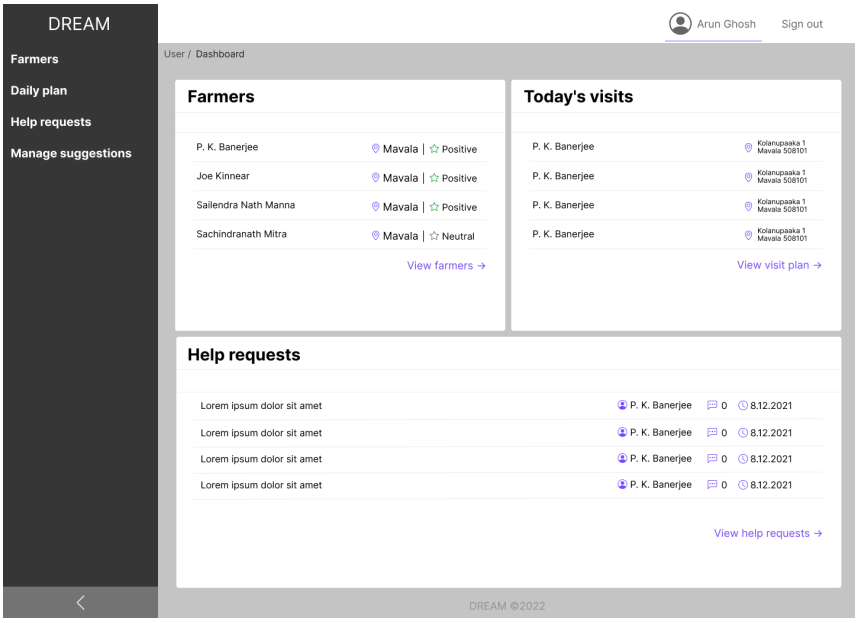


Figure 3.19: **M18.** Agronomist's dashboard.

The agronomist can display a list of farmers in his area of responsibility, using the *Farmer* option in the left navigation bar. The list consists of a table with entries, presenting the most important information about a farmer, as well as a filter bar, allowing the user to search by farmer's name, mandal or note. By clicking on an entry, the user can view a farmer summary analogical to the one available for farmer. This view is identical to the one available for policy maker, using the *Farmer* option in the left navigation bar. The only difference is that policy maker's view contains a list of all farmers in Telangana.

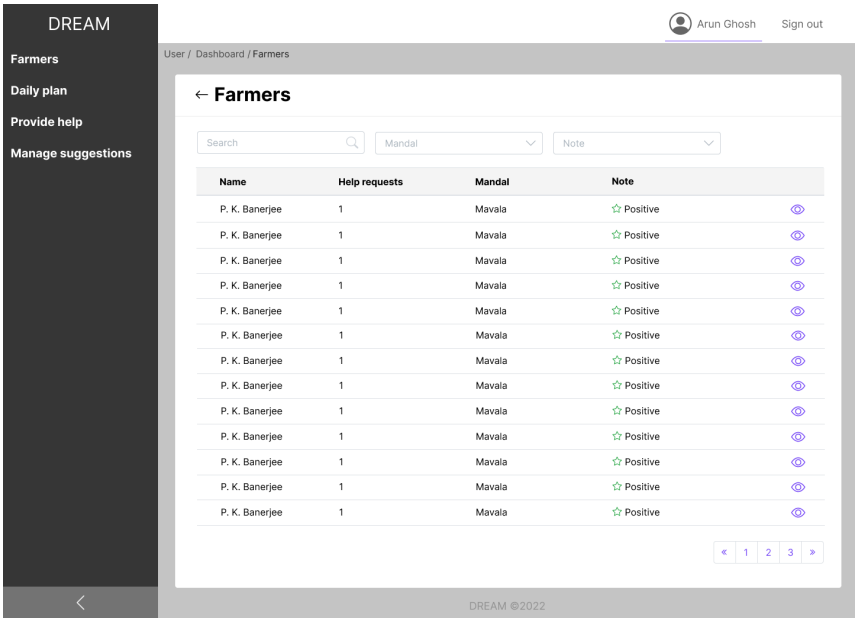


Figure 3.20: **M19.** List of farmers (applicable also for the policy maker).

By clicking on the *Daily plan* button, the agronomist can display his visit calendar and plan for the chosen day. The agronomist can see if he has submitted execution state for the given day: if the day has a green dot on the calendar, it means that the agronomist has realized all his planned visits. A red dot means, that the plan has been submitted, but not all visits were realized. No dot means, that the plan hasn't been submitted yet. The user can manage future visits according to the requirements, using edit and delete buttons next to the given visit. The user can also plan future visits, by clicking on the add button in the upper part of the daily plan.

The described functionality is presented on two following mock-ups: first one shows today/a day in the past, and the next one a day in the future.

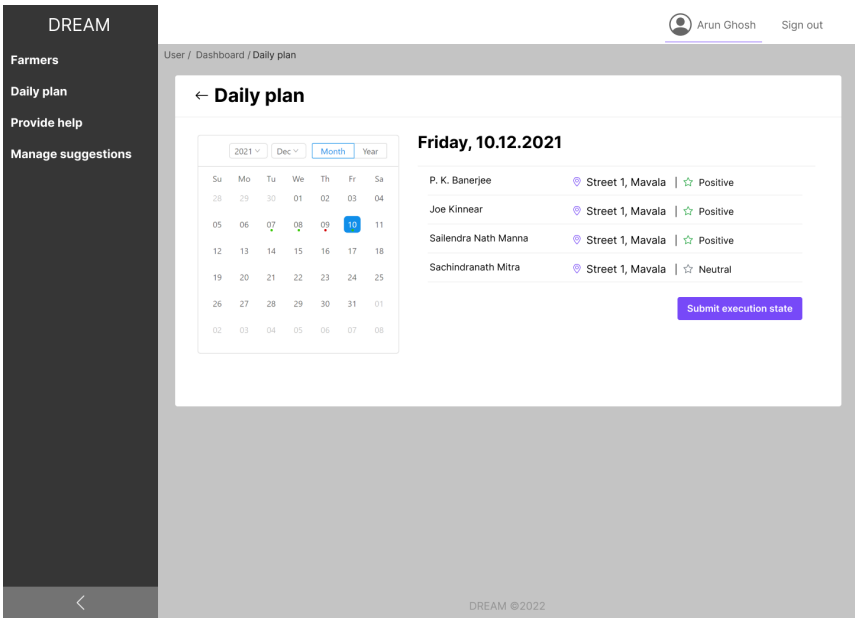


Figure 3.21: **M20.** Calendar of daily plans – view of today or day in the past.

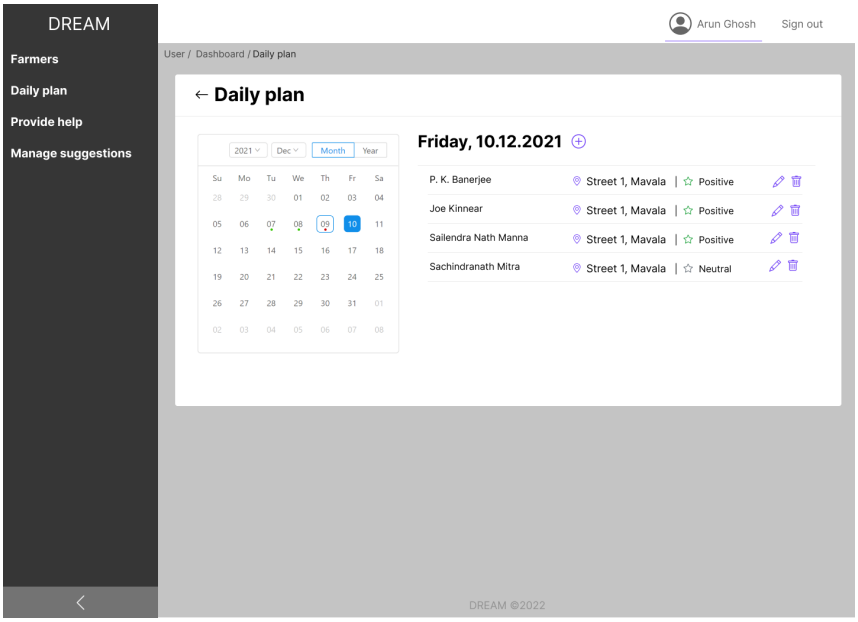


Figure 3.22: **M21.** Calendar of daily plans – view of future day.

After clicking on the *Submit execution state* button on the daily plan of today/past day, a modal dialog appears, allowing the agronomist to select, which farm he has visited and provide comments for farms he hasn't visited.

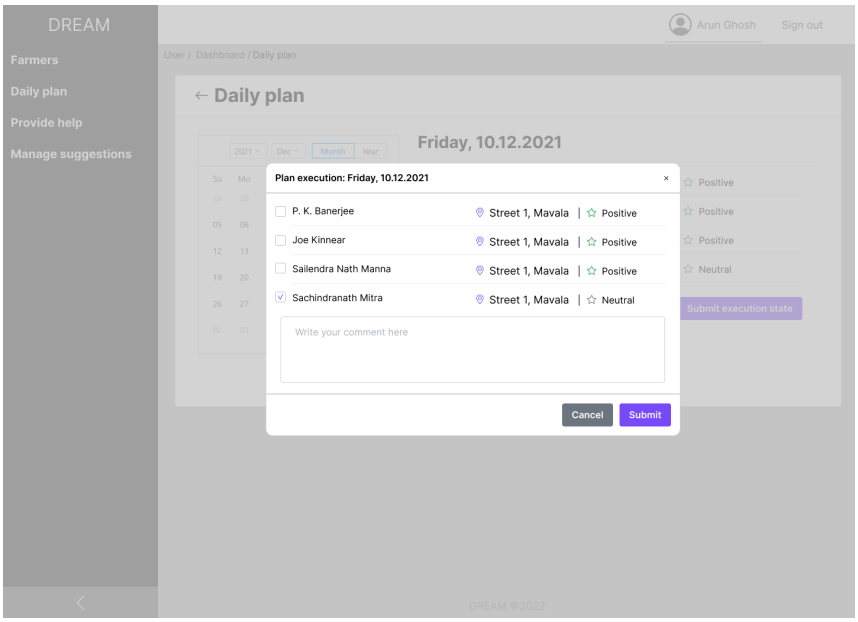


Figure 3.23: **M22.** Submitting executions state of a daily plan.

The agronomist can view all received help request, by clicking on the *Provide help* button in the left navigation bar. This view works exactly the same as the view available for farmers with positive note, presented in figure 3.13.

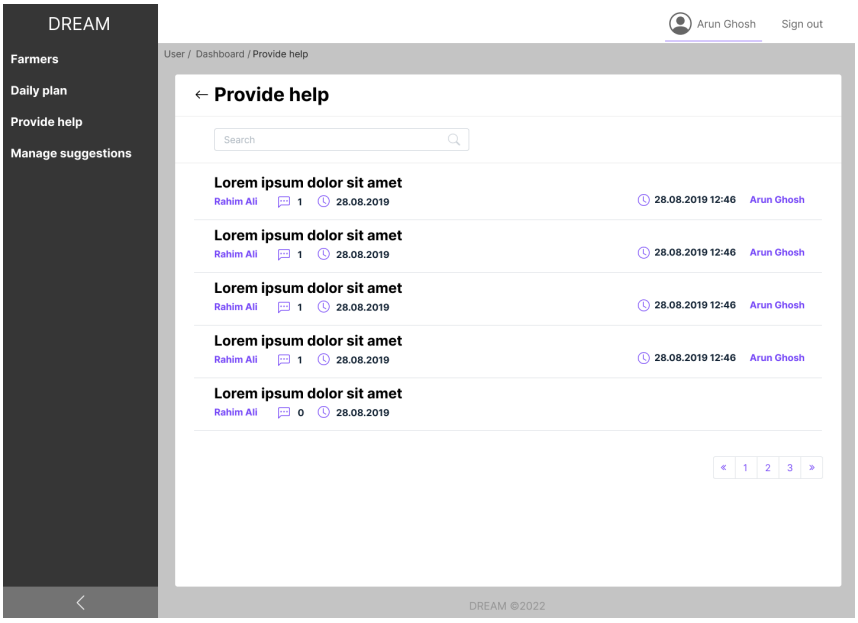


Figure 3.24: **M23.** Help requests received by agronomist.

The agronomist can manage suggestions available to farmers, by clicking on the *Manage suggestions* button in the left navigation bar. Suggestions are presented in a table, containing mandals, corresponding production type as well as the suggestion itself. They can be edited, deleted, and added, using buttons next to entries or *Add new* button in the lower part of the content card. During creation of the suggestion, the user can also choose types of weather, to which the suggestion corresponds.

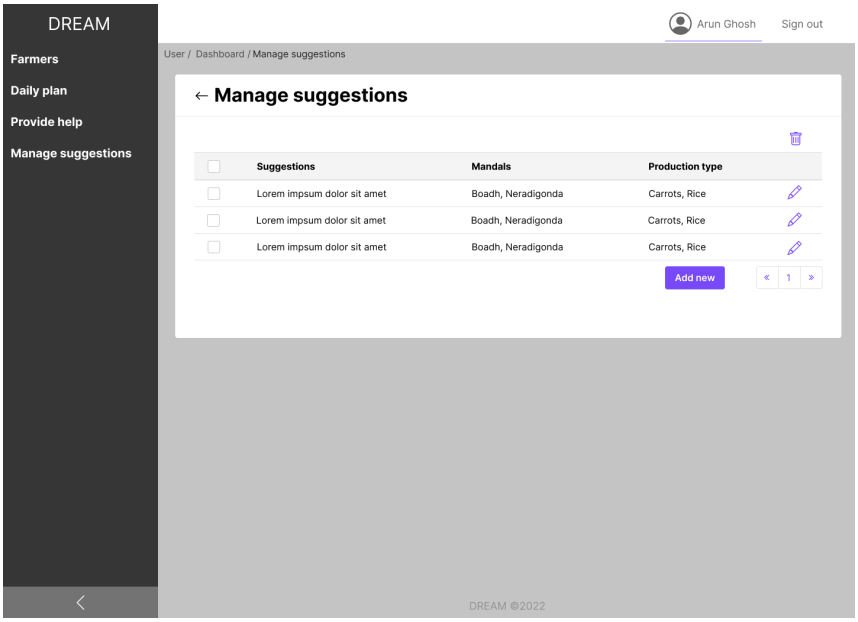


Figure 3.25: **M24.** Managing suggestions.

### Policy maker

The policy maker can view his account summary, by clicking on the button in the top navigation bar, containing his name and surname. In the user view, he can view his account details and delete his account.

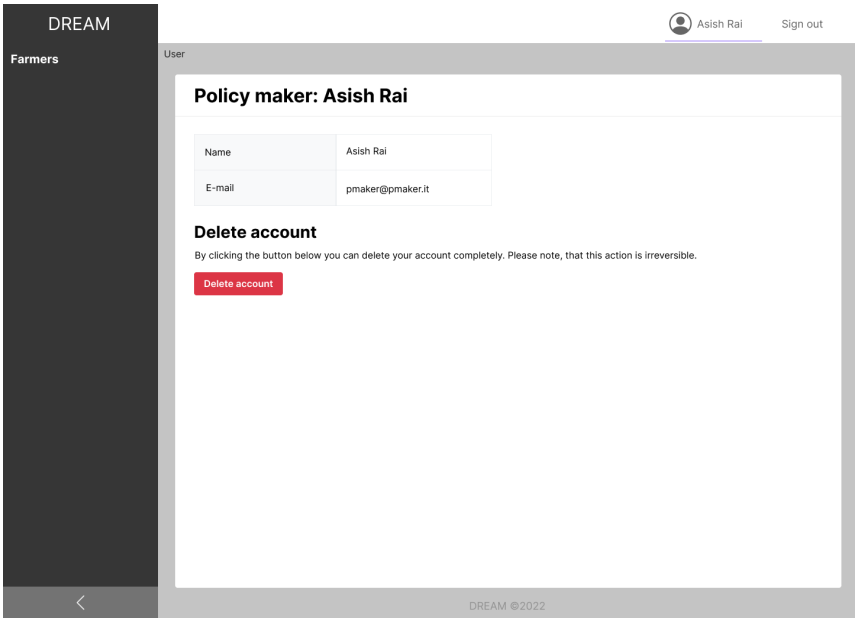


Figure 3.26: **M25.** Policy maker's user view.

Directly after logging in, the policy maker is redirected to the dashboard view, which contains a summary of the well and bad prospering farmers in Telangana.

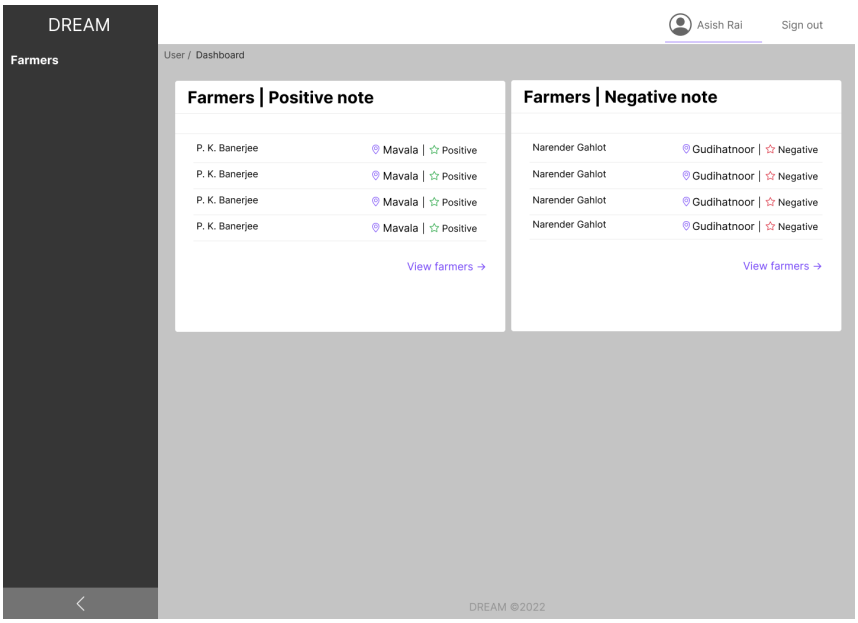


Figure 3.27: **M26.** Policy maker's dashboard.

The policy maker can click on a farmer's entry in the farmer list, shown in figure 3.20. The following farmer's summary is identical to the one described in figure 3.6, with one difference – it allows the policy maker to assign a note, by clicking on the *Change* button in the account details.

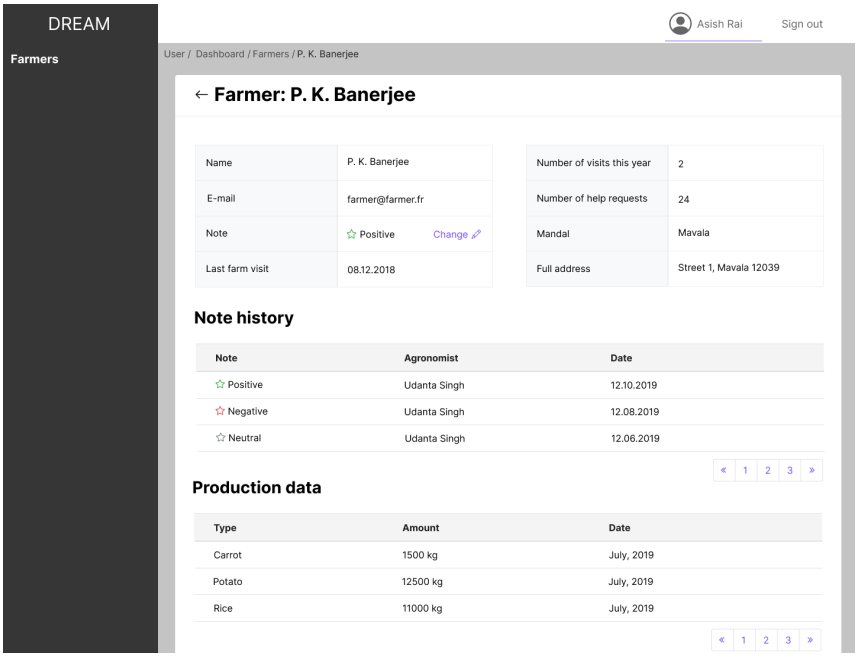


Figure 3.28: **M27.** Partial view of farmer's summary (applicable also to agronomist, without option to change note)

After clicking on the *Change* button, a modal dialog appears. It allows the policy maker to choose a note and a problem type. In case of a negative note, a warning appears, informing the user of consequences of creating a negative note.

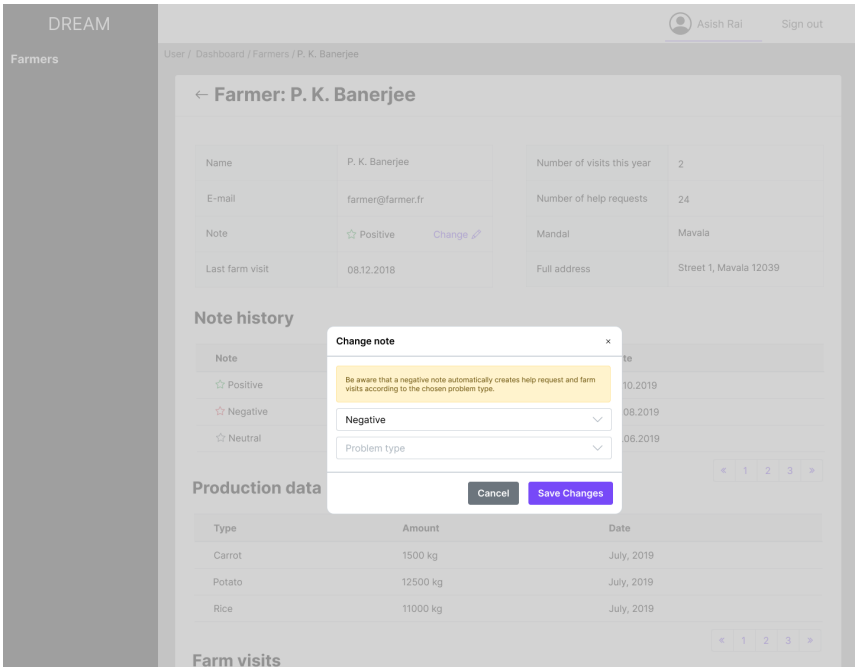


Figure 3.29: **M28.** Assigning note to farmer.

# Chapter 4

# Requirements Traceability

This chapter presents all the requirements defined in RASD together with their mappings onto the system's components defined in the chapter 2. Additionally, a traceability matrix between the requirements and the previously defined mockups is provided.

## 4.1 Functional Requirements

| ID | Requirement |
|---|---|
| **R1.** | The system must uniquely identify each user by his e-mail. |
| **R2.** | The system must allow an unregistered user to create an account with a chosen role. |
| **R3.** | The system must ensure that an agronomist chooses the area of responsibility during the registration process. |
| **R4.** | The system must ensure that a farmer inserts his farm data during the registration process. |
| **R5.** | The system must ensure that two casual farm visits are scheduled for each year of the farm's existence. |
| **R6.** | The system must allow a registered user to log in to the application. |
| **R7.** | The system must allow a registered user to reset his password. |
| **R8.** | The system must allow a logged-in user to sign out of the application. |
| **R9.** | The system must allow a registered user to delete his account. |
| **R10.** | The system must allow a policy maker to assign a note to a farmer. |
| **R11.** | The system must ensure that every farmer initially has a neutral note. |
| **R12.** | The system must have a list of predefined farmer's problem types. |
| **R13.** | The system must allow a policy maker to specify a problem type when assigning a negative note. |

**R14.** The system must ensure that a farm is visited more often in the event of any problems.

**R15.** The system must have a specified number of additional visits caused by each predefined problem type.

**R16.** The system must ensure that only the most recently specified problem type is taken into account when determining the number of visits.

**R17.** The system must ensure that an automatic help request followed by additional farm visits are created, if a farmer receives a negative note.

**R18.** The system must ensure that all the additional farm visits created due to obtaining a negative note are deleted after a farmer's negative note is updated with a positive or a neutral one.

**R19.** The system must allow a policy maker to see a list of all farmers in Telangana.

**R20.** The system must allow a policy maker to see a list of all farmers with a specific note.

**R21.** The system must allow a policy maker to see a list of all farmers in a given mandal.

**R22.** The system must allow a policy maker to find a farmer's summary by his name and surname.

**R23.** The system must allow a policy maker to see a farmer's summary.

**R24.** The system must allow a farmer to see his own farmer's summary.

**R25.** The system must allow a farmer to see personalized suggestions.

**R26.** The system must allow a farmer to manage his monthly production data.

**R27.** The system must allow a farmer to see a list of help requests created by him.

**R28.** The system must allow a farmer to find a help request created by him by the topic.

**R29.** The system must allow a farmer to see a specific help request created by him.

**R30.** The system must allow a farmer to delete a specific help request created by him.

**R31.** The system must allow a farmer to create a new help request.

**R32.** The system must allow a farmer with a positive note to see a list of all help requests in his mandal.

**R33.** The system must allow a farmer with a positive note to find a help request in his mandal by the topic.

**R34.** The system must allow a farmer with a positive note to see a specific help request in his mandal.

**R35.** The system must allow a farmer with a positive note to respond to a specific help request in his mandal.

**R36.** The system must allow a farmer with a positive note to delete a help response, only if it was created by him.

**R37.** The farmer is able to see farmer's summary of a farmer with a negative note whose help request he received.

**R38.** The system must allow a farmer to see a list of all forum threads in Telangana.

**R39.** The system must allow a farmer to find a forum thread by the topic.

**R40.** The system must allow a farmer to see a specific forum thread with its comments.

**R41.** The system must allow a farmer to create a comment in a forum thread.

**R42.** The system must allow a farmer to delete a comment in a forum thread, only if it was created by him.

**R43.** The system must allow a farmer to create a forum thread.

**R44.** The system must allow an agronomist to manage his area of responsibility.

**R45.** The system must allow an agronomist to see the list of suggestions for mandals in his area of responsibility.

**R46.** The system must allow an agronomist to manage the list of suggestions for mandals in his area of responsibility.

**R47.** The system must allow an agronomist to see a list of all farmers in his area of responsibility.

**R48.** The system must allow an agronomist to see a list of all farmers with a specific note in his area of responsibility.

**R49.** The system must allow an agronomist to see a list of all farmers in a given mandal in his area of responsibility.

**R50.** The system must allow an agronomist to find a farmer's summary in his area of responsibility by his name and surname.

**R51.** The system must allow an agronomist to see a farmer's summary in his area of responsibility.

**R52.** The system must allow an agronomist to see a list of all help requests in his area of responsibility.

**R53.** The system must allow an agronomist to find a help request in his area of responsibility by the topic.

**R54.** The system must allow an agronomist to see a specific help request in his area of responsibility.

**R55.** The system must allow an agronomist to respond to a specific help request in his area of responsibility.

**R56.** The system must allow an agronomist to delete a help response, only if it was created by him.

**R57.** The system must allow an agronomist to see his daily plans.

**R58.** The system must allow an agronomist to submit a daily plan's execution state, by rejecting or confirming each visit, on the same date or after the daily plan's date has passed.

**R59.** The system must allow an agronomist to provide a comment for a visit he is confirming.

**R60.** The system must ensure that after an agronomist submits his daily plan, for all the causal visits marked as confirmed, new ones are created in approximately half of a year.

**R61.** The system must allow an agronomist to delete a visit before its date.

| | |
|---|---|
| **R62.** | The system must ensure that a deleted visit is marked as rejected. |
| **R63.** | The system must ensure that in case of rejecting a casual visit, a new one is created in maximally 5 days. |
| **R64.** | The system must allow replanning any different visit than a casual one without any constraints. |
| **R65.** | The system must read and update weather forecasts every day. |
| **R66.** | The system must read and store data from humidity sensors every day. |
| **R67.** | The system must read and store data from water irrigation systems every day. |
| **R68.** | The system must ensure that a farmer who creates a help request cannot be its recipient. |

## 4.2 Mapping between system's components and functional requirements

| | |
|---|---|
| **G1** | Improve farmers performance by providing them with personalized suggestions. |
| **R1.** | The system must uniquely identify each user by his e-mail.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R2.** | The system must allow an unregistered user to create an account with a chosen role.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R3.** | The system must ensure that an agronomist chooses the area of responsibility during the registration process.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R4.** | The system must ensure that a farmer inserts his farm data during the registration process.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R6.** | The system must allow a registered user to log in to the application.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R7.** | The system must allow a registered user to reset his password.<br>(**WebApplication**, **AccountController**, **AccountService**, **EmailSender**, **DataAccess**) |
| **R8.** | The system must allow a logged-in user to sign out of the application.<br>(**WebApplication**) |
| **R9.** | The system must allow a registered user to delete his account.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R25.** | The system must allow a farmer to see personalized suggestions.<br>(**WebApplication**, **SuggestionController**, **SuggestionService**, **DataAccess**) |

| | |
|---|---|
| **R26.** | The system must allow a farmer to manage his monthly production data.<br>(**WebApplication**, **FarmerController**, **FarmService**, **DataAccess**) |
| **R44.** | The system must allow an agronomist to manage his area of responsibility.<br>(**WebApplication**, **AgronomistController**, **AgronomistService**, **DataAccess**) |
| **R45.** | The system must allow an agronomist to see the list of suggestions for mandals in his area of responsibility.<br>(**WebApplication**, **SuggestionController**, **SuggestionService**, **DataAccess**) |
| **R46.** | The system must allow an agronomist to manage the list of suggestions for mandals in his area of responsibility.<br>(**WebApplication**, **SuggestionsController**, **SuggestionsService**, **DataAccess**) |

Table 4.2: G1 mapping onto components and requirements.

| **G2** | Acquire, combine, and visualize data from external systems. |
|---|---|
| **R1.** | The system must uniquely identify each user by his e-mail.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R2.** | The system must allow an unregistered user to create an account with a chosen role.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R3.** | The system must ensure that an agronomist chooses the area of responsibility during the registration process.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R4.** | The system must ensure that a farmer inserts his farm data during the registration process.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R5.** | The system must ensure that two casual farm visits are scheduled for each year of the farm's existence.<br>(**WebApplication**, **AccountController**, **AccountService**, **VisitsPlanner**, **DataAccess**) |
| **R6.** | The system must allow a registered user to log in to the application.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R7.** | The system must allow a registered user to reset his password.<br>(**WebApplication**, **AccountController**, **AccountService**, **EmailSender**, **DataAccess**) |
| **R8.** | The system must allow a logged-in user to sign out of the application.<br>(**WebApplication**) |
| **R9.** | The system must allow a registered user to delete his account.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |

| | |
|---|---|
| **R23.** | The system must allow a policy maker to see a farmer's summary.<br>(**WebApplication**, **FarmerController**, **FarmerService**, **FarmService**, **DataAccess**) |
| **R24.** | The system must allow a farmer to see his own farmer's summary.<br>(**WebApplication**, **FarmerController**, **FarmerService**, **FarmService**, **DataAccess**) |
| **R37.** | The farmer is able to see farmer's summary of a farmer with a negative note whose help request he received.<br>(**WebApplication**, **FarmerController**, **FarmerService**, **FarmService**, **DataAccess**) |
| **R51.** | The system must allow an agronomist to see a farmer's summary in his area of responsibility.<br>(**WebApplication**, **FarmerController**, **FarmerService**, **FarmService**, **DataAccess**) |
| **R65.** | The system must read and update weather forecasts every day.<br>(**WeatherForecastController**, **WeatherForecastService**, **ExternalSystemsReader**, **DataAccess**) |
| **R66.** | The system must read and store data from humidity sensors every day.<br>(**FarmerController**, **FarmerService**, **FarmService**, **ExternalSystemsReader**, **DataAccess**) |
| **R67.** | The system must read and store data from water irrigation systems every day.<br>(**FarmerController**, **FarmerService**, **FarmService**, **ExternalSystemsReader**, **DataAccess**) |

Table 4.3: G2 mapping onto components and requirements.

| | |
|---|---|
| **G3** | Facilitate performance assessment of the farmers. |
| **R1.** | The system must uniquely identify each user by his e-mail.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R2.** | The system must allow an unregistered user to create an account with a chosen role.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R3.** | The system must ensure that an agronomist chooses the area of responsibility during the registration process.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R4.** | The system must ensure that a farmer inserts his farm data during the registration process.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |

| | |
|---|---|
| **R5.** | The system must ensure that two casual farm visits are scheduled for each year of the farm's existence.<br>(**WebApplication**, **AccountController**, **AccountService**, **VisitsPlanner**, **DataAccess**) |
| **R6.** | The system must allow a registered user to log in to the application.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R7.** | The system must allow a registered user to reset his password.<br>(**WebApplication**, **AccountController**, **AccountService**, **EmailSender**, **DataAccess**) |
| **R8.** | The system must allow a logged-in user to sign out of the application.<br>(**WebApplication**) |
| **R9.** | The system must allow a registered user to delete his account.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R10.** | The system must allow a policy maker to assign a note to a farmer.<br>(**WebApplication**, **FarmerController**, **FarmerService**, **DataAccess**) |
| **R11.** | The system must ensure that every farmer initially has a neutral note.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R12.** | The system must have a list of predefined farmer's problem types.<br>(**WebApplication**, **FarmerController**, **FarmerService**, **DataAccess**) |
| **R13.** | The system must allow a policy maker to specify a problem type when assigning a negative note.<br>(**WebApplication**, **FarmerController**, **FarmerService**, **DataAccess**) |
| **R19.** | The system must allow a policy maker to see a list of all farmers in Telangana.<br>(**WebApplication**, **FarmerController**, **FarmerService**, **DataAccess**) |
| **R20.** | The system must allow a policy maker to see a list of all farmers with a specific note.<br>(**WebApplication**, **FarmerController**, **FarmerService**, **DataAccess**) |
| **R21.** | The system must allow a policy maker to see a list of all farmers in a given mandal.<br>(**WebApplication**, **FarmerController**, **FarmerService**, **DataAccess**) |
| **R22.** | The system must allow a policy maker to find a farmer's summary by his name and surname.<br>(**WebApplication**, **FarmerController**, **FarmerService**, **DataAccess**) |
| **R23.** | The system must allow a policy maker to see a farmer's summary.<br>(**WebApplication**, **FarmerController**, **FarmerService**, **FarmService**, **DataAccess**) |

Table 4.4: G3 mapping onto components and requirements.

| **G4** | Promote regular farms' visits by agronomists. |
|---|---|
| **R1.** | The system must uniquely identify each user by his e-mail.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R2.** | The system must allow an unregistered user to create an account with a chosen role.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R3.** | The system must ensure that an agronomist chooses the area of responsibility during the registration process.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R4.** | The system must ensure that a farmer inserts his farm data during the registration process.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R5.** | The system must ensure that two casual farm visits are scheduled for each year of the farm's existence.<br>(**WebApplication**, **AccountController**, **AccountService**, **VisitsPlanner**, **DataAccess**) |
| **R6.** | The system must allow a registered user to log in to the application.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R7.** | The system must allow a registered user to reset his password.<br>(**WebApplication**, **AccountController**, **AccountService**, **EmailSender**, **DataAccess**) |
| **R8.** | The system must allow a logged-in user to sign out of the application.<br>(**WebApplication**) |
| **R9.** | The system must allow a registered user to delete his account.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R10.** | The system must allow a policy maker to assign a note to a farmer.<br>(**WebApplication**, **FarmerController**, **FarmerService**, **DataAccess**) |
| **R11.** | The system must ensure that every farmer initially has a neutral note.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R12.** | The system must have a list of predefined farmer's problem types.<br>(**WebApplication**, **FarmerController**, **FarmerService**, **DataAccess**) |
| **R13.** | The system must allow a policy maker to specify a problem type when assigning a negative note.<br>(**WebApplication**, **FarmerController**, **FarmerService**, **DataAccess**) |
| **R14.** | The system must ensure that a farm is visited more often in the event of any problems. |

(**WebApplication**, **AgronomistController**, **AgronomistService**, **VisitsPlanner**, **DataAccess**)

**R15.** The system must have a specified number of additional visits caused by each predefined problem type.

(**WebApplication**, **AgronomistController**, **AgronomistService**, **VisitsPlanner**, **DataAccess**)

**R16.** The system must ensure that only the most recently specified problem type is taken into account when determining the number of visits.

(**WebApplication**, **AgronomistController**, **AgronomistService**, **VisitsPlanner**, **DataAccess**)

**R17.** The system must ensure that an automatic help request followed by additional farm visits are created, if a farmer receives a negative note.

(**WebApplication**, **RequestController**, **RequestService**, **VisitsPlanner**, **DataAccess**)

**R18.** The system must ensure that all the additional farm visits created due to obtaining a negative note are deleted after a farmer's negative note is updated with a positive or a neutral one.

(**WebApplication**, **AgronomistController**, **AgronomistService**, **VisitsPlanner**, **DataAccess**)

**R44.** The system must allow an agronomist to manage his area of responsibility.

(**WebApplication**, **AgronomistController**, **AgronomistService**, **DataAccess**)

**R57.** The system must allow an agronomist to see his daily plans.

(**WebApplication**, **AgronomistController**, **AgronomistService**, **DataAccess**)

**R58.** The system must allow an agronomist to submit a daily plan's execution state, by rejecting or confirming each visit, on the same date or after the daily plan's date has passed.

(**WebApplication**, **AgronomistController**, **AgronomistService**, **VisitsPlanner**, **DataAccess**)

**R59.** The system must allow an agronomist to provide a comment for a visit he is confirming.

(**WebApplication**, **AgronomistController**, **AgronomistService**, **DataAccess**)

**R60.** The system must ensure that after an agronomist submits his daily plan, for all the causal visits marked as confirmed, new ones are created in approximately half of a year.

(**WebApplication**, **AgronomistController**, **AgronomistService**, **VisitsPlanner**, **DataAccess**)

**R61.** The system must allow an agronomist to delete a visit before its date.

(**WebApplication**, **AgronomistController**, **AgronomistService**, **VisitsPlanner**, **DataAccess**)

**R62.** The system must ensure that a deleted visit is marked as rejected.

| | |
|---|---|
| | (**AgronomistController**, **AgronomistService**, **DataAccess**) |
| **R63.** | The system must ensure that in case of rejecting a casual visit, a new one is created in maximally 5 days.<br>(**WebApplication**, **AgronomistController**, **AgronomistService**, **VisitsPlanner**, **DataAccess**) |
| **R64.** | The system must allow replanning any different visit than a casual one without any constraints.<br>(**WebApplication**, **AgronomistController**, **AgronomistService**, **VisitsPlanner**, **DataAccess**) |

Table 4.5: G4 mapping onto components and requirements.

| **G5** | Enable agronomists to exchange information with farmers. |
|---|---|
| **R1.** | The system must uniquely identify each user by his e-mail.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R2.** | The system must allow an unregistered user to create an account with a chosen role.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R3.** | The system must ensure that an agronomist chooses the area of responsibility during the registration process.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R4.** | The system must ensure that a farmer inserts his farm data during the registration process.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R5.** | The system must ensure that two casual farm visits are scheduled for each year of the farm's existence.<br>(**WebApplication**, **AccountController**, **AccountService**, **VisitsPlanner**, **DataAccess**) |
| **R6.** | The system must allow a registered user to log in to the application.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R7.** | The system must allow a registered user to reset his password.<br>(**WebApplication**, **AccountController**, **AccountService**, **EmailSender**, **DataAccess**) |
| **R8.** | The system must allow a logged-in user to sign out of the application.<br>(**WebApplication**) |
| **R9.** | The system must allow a registered user to delete his account.<br>(**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R17.** | The system must ensure that an automatic help request followed by additional farm visits are created, if a farmer receives a negative note. |

(**WebApplication**, **RequestController**, **RequestService**, **VisitsPlanner**, **DataAccess**)

**R27.** The system must allow a farmer to see a list of help requests created by him.

(**WebApplication**, **RequestController**, **RequestService**, **DataAccess**)

**R28.** The system must allow a farmer to find a help request created by him by the topic.

(**WebApplication**, **RequestController**, **RequestService**, **DataAccess**)

**R29.** The system must allow a farmer to see a specific help request created by him.

(**WebApplication**, **RequestController**, **RequestService**, **DataAccess**)

**R30.** The system must allow a farmer to delete a specific help request created by him.

(**WebApplication**, **RequestController**, **RequestService**, **DataAccess**)

**R31.** The system must allow a farmer to create a new help request.

(**WebApplication**, **RequestController**, **RequestService**, **DataAccess**)

**R44.** The system must allow an agronomist to manage his area of responsibility.

(**WebApplication**, **AgronomistController**, **AgronomistService**, **DataAccess**)

**R47.** The system must allow an agronomist to see a list of all farmers in his area of responsibility.

(**WebApplication**, **FarmerController**, **FarmerService**, **DataAccess**)

**R48.** The system must allow an agronomist to see a list of all farmers with a specific note in his area of responsibility.

(**WebApplication**, **FarmerController**, **FarmertService**, **DataAccess**)

**R49.** The system must allow an agronomist to see a list of all farmers in a given mandal in his area of responsibility.

(**WebApplication**, **FarmerController**, **FarmertService**, **DataAccess**)

**R50.** The system must allow an agronomist to find a farmer's summary in his area of responsibility by his name and surname.

(**WebApplication**, **FarmerController**, **FarmerService**, **FarmService**, **DataAccess**)

**R51.** The system must allow an agronomist to see a farmer's summary in his area of responsibility.

(**WebApplication**, **FarmerController**, **FarmerService**, **FarmService**, **DataAccess**)

**R52.** The system must allow an agronomist to see a list of all help requests in his area of responsibility.

(**WebApplication**, **RequestController**, **RequestService**, **DataAccess**)

**R53.** The system must allow an agronomist to find a help request in his area of responsibility by the topic.

(**WebApplication**, **RequestController**, **RequestService**, **DataAccess**)

**R54.** The system must allow an agronomist to see a specific help request in his area of responsibility.

| | |
|---|---|
| | (**WebApplication**, **RequestController**, **RequestService**, **DataAccess**) |
| **R55.** | The system must allow an agronomist to respond to a specific help request in his area of responsibility. |
| | (**WebApplication**, **RequestController**, **RequestService**, **DataAccess**) |
| **R56.** | The system must allow an agronomist to delete a help response, only if it was created by him. |
| | (**WebApplication**, **RequestController**, **RequestService**, **DataAccess**) |

Table 4.6: G5 mapping onto components and requirements.

| **G6** | Enable farmers to exchange their knowledge. |
|---|---|
| **R1.** | The system must uniquely identify each user by his e-mail. |
| | (**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R2.** | The system must allow an unregistered user to create an account with a chosen role. |
| | (**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R3.** | The system must ensure that an agronomist chooses the area of responsibility during the registration process. |
| | (**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R4.** | The system must ensure that a farmer inserts his farm data during the registration process. |
| | (**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R5.** | The system must ensure that two casual farm visits are scheduled for each year of the farm's existence. |
| | (**WebApplication**, **AccountController**, **AccountService**, **VisitsPlanner**, **DataAccess**) |
| **R6.** | The system must allow a registered user to log in to the application. |
| | (**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R7.** | The system must allow a registered user to reset his password. |
| | (**WebApplication**, **AccountController**, **AccountService**, **EmailSender**, **DataAccess**) |
| **R8.** | The system must allow a logged-in user to sign out of the application. |
| | (**WebApplication**) |
| **R9.** | The system must allow a registered user to delete his account. |
| | (**WebApplication**, **AccountController**, **AccountService**, **DataAccess**) |
| **R17.** | The system must ensure that an automatic help request followed by additional farm visits are created, if a farmer receives a negative note. |

(**WebApplication**, **RequestController**, **RequestService**, **VisitsPlanner**, **DataAccess**)

**R27.** The system must allow a farmer to see a list of help requests created by him.

(**WebApplication**, **RequestController**, **RequestService**, **DataAccess**)

**R28.** The system must allow a farmer to find a help request created by him by the topic.

(**WebApplication**, **RequestController**, **RequestService**, **DataAccess**)

**R29.** The system must allow a farmer to see a specific help request created by him.

(**WebApplication**, **RequestController**, **RequestService**, **DataAccess**)

**R30.** The system must allow a farmer to delete a specific help request created by him.

(**WebApplication**, **RequestController**, **RequestService**, **DataAccess**)

**R31.** The system must allow a farmer to create a new help request.

(**WebApplication**, **RequestController**, **RequestService**, **DataAccess**)

**R32.** The system must allow a farmer with a positive note to see a list of all help requests in his mandal.

(**WebApplication**, **RequestController**, **RequestService**, **DataAccess**)

**R33.** The system must allow a farmer with a positive note to find a help request in his mandal by the topic.

(**WebApplication**, **RequestController**, **RequestService**, **DataAccess**)

**R34.** The system must allow a farmer with a positive note to see a specific help request in his mandal.

(**WebApplication**, **RequestController**, **RequestService**, **DataAccess**)

**R35.** The system must allow a farmer with a positive note to respond to a specific help request in his mandal.

(**WebApplication**, **RequestController**, **RequestService**, **DataAccess**)

**R36.** The system must allow a farmer with a positive note to delete a help response, only if it was created by him.

(**WebApplication**, **RequestController**, **RequestService**, **DataAccess**)

**R37.** The farmer is able to see farmer's summary of a farmer with a negative note whose help request he received.

(**WebApplication**, **FarmerController**, **FarmerService**, **FarmService**, **DataAccess**)

**R38.** The system must allow a farmer to see a list of all forum threads in Telangana.

(**WebApplication**, **ForumController**, **ForumService**, **DataAccess**)

**R39.** The system must allow a farmer to find a forum thread by the topic.

(**WebApplication**, **ForumController**, **ForumService**, **DataAccess**)

**R40.** The system must allow a farmer to see a specific forum thread with its comments.

(**WebApplication**, **ForumController**, **ForumService**, **DataAccess**)

**R41.** The system must allow a farmer to create a comment in a forum thread.

(**WebApplication**, **ForumController**, **ForumService**, **DataAccess**)

**R42.** The system must allow a farmer to delete a comment in a forum thread, only if it was created by him.

(**WebApplication**, **ForumController**, **ForumService**, **DataAccess**)

**R43.** The system must allow a farmer to create a forum thread.

(**WebApplication**, **ForumController**, **ForumService**, **DataAccess**)

**R68.** The system must ensure that a farmer who creates a help request cannot be its recipient.

(**ForumController**, **ForumService**, **DataAccess**)

Table 4.7: G6 mapping onto components and requirements.

## 4.3 Mapping between functional requirements and mockups

Table 4.8 depicts a mapping of system mockups onto the functional requirements. Not all the requirements could be mapped, since some of them are handled by the backend server and are not visible on the frontend side.

| | M1 | M2 | M3 | M4 | M5 | M6 | M7 | M8 | M9 | M10 | M11 | M12 | M13 | M14 | M15 | M16 | M17 | M18 | M19 | M20 | M21 | M22 | M23 | M24 | M25 | M26 | M27 | M28 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R2 | ✔ | ✔ | ✔ | | | | | | | | | | | | | | | | | | | | | | | | | |
| R3 | | | ✔ | | | | | | | | | | | | | | | | | | | | | | | | | |
| R4 | | ✔ | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R5 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R6 | | | | ✔ | | | | | | | | | | | | | | | | | | | | | | | | |
| R7 | | | | | ✔ | | | | | | | | | | | | | | | | | | | | | | | |
| R8 | | | | | | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |
| R9 | | | | | | ✔ | | | | | | | | | | | ✔ | | | | | | | | ✔ | | | |
| R10 | | | | | | | | | | | | | | | | | | | | | | | | | | | ✔ | ✔ |
| R11 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R12 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R13 | | | | | | | | | | | | | | | | | | | | | | | | | | | | ✔ |
| R14 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R15 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R16 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R17 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R18 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R19 | | | | | | | | | | | | | | | | | | | ✔ | | | | | | | | | |
| R20 | | | | | | | | | | | | | | | | | | | ✔ | | | | | | | | | |
| R21 | | | | | | | | | | | | | | | | | | | ✔ | | | | | | | | | |
| R22 | | | | | | | | | | | | | | | | | | | ✔ | | | | | | | | | |
| R23 | | | | | | | | | | | | | | | | | | | | | | | | | | | ✔ | |
| R24 | | | | | | ✔ | | | | | | | | | | | | | | | | | | | | | | |
| R25 | | | | | | | ✔ | | | | | | | | | | | | | | | | | | | | | |
| R26 | | | | | | | | ✔ | | | | | | | | | | | | | | | | | | | | |
| R27 | | | | | | | | | ✔ | | | | | | | | | | | | | | | | | | | |
| R28 | | | | | | | | | ✔ | | | | | | | | | | | | | | | | | | | |
| R29 | | | | | | | | | | ✔ | | | | | | | | | | | | | | | | | | |
| R30 | | | | | | | | | | ✔ | | | | | | | | | | | | | | | | | | |
| R31 | | | | | | | | | | | ✔ | | | | | | | | | | | | | | | | | |
| R32 | | | | | | | | | | | | ✔ | | | | | | | | | | | | | | | | |
| R33 | | | | | | | | | | | | ✔ | | | | | | | | | | | | | | | | |

| | M1 | M2 | M3 | M4 | M5 | M6 | M7 | M8 | M9 | M10 | M11 | M12 | M13 | M14 | M15 | M16 | M17 | M18 | M19 | M20 | M21 | M22 | M23 | M24 | M25 | M26 | M27 | M28 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R34 | | | | | | | | | | | | | ✔ | | | | | | | | | | | | | | | |
| R35 | | | | | | | | | | | | | ✔ | | | | | | | | | | | | | | | |
| R36 | | | | | | | | | | | | | ✔ | | | | | | | | | | | | | | | |
| R37 | | | | | | | | | | | | | ✔ | | | | | | | | | | | | | | | |
| R38 | | | | | | | | | | | | | | ✔ | | | | | | | | | | | | | | |
| R39 | | | | | | | | | | | | | | ✔ | | | | | | | | | | | | | | |
| R40 | | | | | | | | | | | | | | | ✔ | | | | | | | | | | | | | |
| R41 | | | | | | | | | | | | | | | ✔ | | | | | | | | | | | | | |
| R42 | | | | | | | | | | | | | | | ✔ | | | | | | | | | | | | | |
| R43 | | | | | | | | | | | | | | | | ✔ | | | | | | | | | | | | |
| R44 | | | | | | | | | | | | | | | | | ✔ | | | | | | | | | | | |
| R45 | | | | | | | | | | | | | | | | | | | | | | | | ✔ | | | | |
| R46 | | | | | | | | | | | | | | | | | | | | | | | | ✔ | | | | |
| R47 | | | | | | | | | | | | | | | | | | | ✔ | | | | | | | | | |
| R48 | | | | | | | | | | | | | | | | | | | ✔ | | | | | | | | | |
| R49 | | | | | | | | | | | | | | | | | | | ✔ | | | | | | | | | |
| R50 | | | | | | | | | | | | | | | | | | | ✔ | | | | | | | | | |
| R51 | | | | | | | | | | | | | | | | | | | | | | | | | | | ✔ | |
| R52 | | | | | | | | | | | | | | | | | | | | | | | ✔ | | | | | |
| R53 | | | | | | | | | | | | | | | | | | | | | | | ✔ | | | | | |
| R54 | | | | | | | | | | | | | ✔ | | | | | | | | | | | | | | | |
| R55 | | | | | | | | | | | | | ✔ | | | | | | | | | | | | | | | |
| R56 | | | | | | | | | | | | | ✔ | | | | | | | | | | | | | | | |
| R57 | | | | | | | | | | | | | | | | | | | | ✔ | ✔ | | | | | | | |
| R58 | | | | | | | | | | | | | | | | | | | | ✔ | | | | | | | | |
| R59 | | | | | | | | | | | | | | | | | | | | | | ✔ | | | | | | |
| R60 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R61 | | | | | | | | | | | | | | | | | | | | | ✔ | | | | | | | |
| R62 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R63 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R64 | | | | | | | | | | | | | | | | | | | | | ✔ | | | | | | | |
| R65 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R66 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R67 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R68 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

Table 4.8: Mockups mapping onto functional requirements.

# Chapter 5

# Implementation, Integration and Test Plan

## 5.1 Implementation plan

Motivated by effectiveness of the agile approach as opposed to the traditional ones, the development of the DREAM system is organized in an iterative way [13]. The stakeholders should assist throughout the whole development process, providing relevant feedback. The goal is to quickly provide the Minimum Viable Product (MVP) that has just enough functionality to satisfy the users and collect feedback for further development as soon as possible [17].

In order to break down the implementation into iterative phases, a list of dependencies between the components was created:

**D1.** The Business Layer components' implementation requires prior implementation of Data Access Layer to provide persistence of data.

**D2.** The implementation of *AccountController* and *AccountService* requires prior implementation of *EmailSender* to provide the functionality of sending reset password e-mails.

**D3.** The implementation of all *controller*s and dependent services requires prior implementation of *AccountController* and *AccountService* to provide the authorization mechanisms required by the HTTP endpoints defined by the *controller*s.

**D4.** The implementation of *AgronomistController* and *AgronomistService* requires prior implementation of *VisitsPlanner* to provide the implementation of Visit Scheduling Algorithm (2.7.1) that ensures each farm is visited at least twice a year.

**D5.** The implementation of *FarmerController* and *FarmerService* requires prior implementation of *VisitsPlanner* to provide the implementation of Visit Scheduling Algorithm (2.7.1) that ensures each farm is visited more frequently if a farmer has obtained a negative note.

**D6.** Optional: The implementation of *FarmerController* and *FarmService* as well as *WeatherForecastController* and *WeatherForecastService* requires prior implementation of *ExternalSystemsReader* to ensure that database contains data from external systems to be read by dependent components. However, this dependency can be relaxed either by accepting empty values returned by the API or by providing some static data.

All components not mentioned above can be built in any sequence. However, a specific implementation order, which favors the delivery of users' most valued functionalities first, is proposed in the following part of the document.

The implementation of the Web Application views presented in the chapter 3 is achieved in an iterative way, by the creation of the views corresponding to the DREAM Server Application functionalities developed in the given iteration. However, the views can be developed independently since the communication with the DREAM Server Application can be mocked.

The first development phase will focus on the Data Access Layer of the DREAM Server Application as well as the components required for users' registration and log in to address the dependencies **D1**, **D2**, and **D3**. The components built in this step are presented in the figure 5.1. In case of any issues, the first phase can be further broken down into two phases, with the first part containing only the Data Access components.

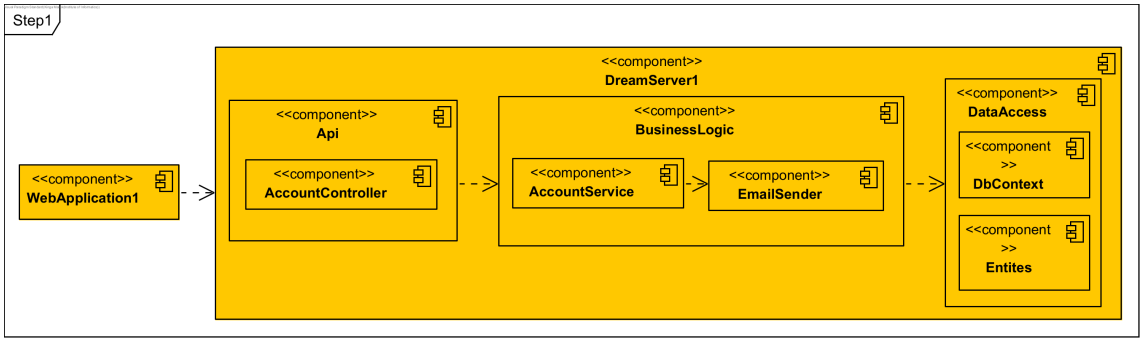


Figure 5.1: Step 1 of the implementation plan. Components added in this step are highlighted with orange color.

The second step of the development is presented in the figure 5.2. The focus of this stage is on delivery of the first functionality available for the farmers after registration in the system. Those farmers will be able to exchange knowledge using the forum. Afterwards, the first MVP for the farmers' can be released. Additionally, this phase aims at providing *VisitsPlanner* to allow the implementation of *AgronomistController* and *AgronomistService* in the following one (resolving the dependency **D4**). Furthermore, the introduction of requests for help functionality is started by development of *RequestService*.

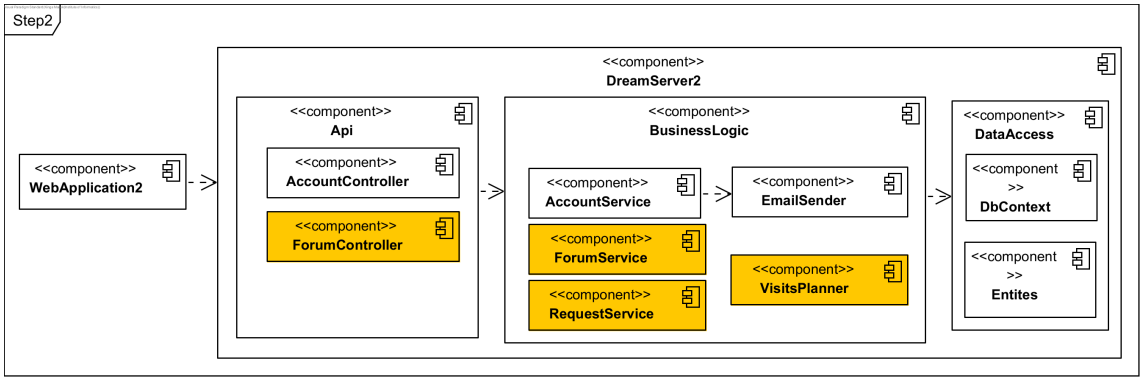


Figure 5.2: Step 2 of the implementation plan. Components added in this step are highlighted with orange color.

Thereafter, the third phase presented in the figure 5.3 is focused on finishing the requests for help functionality. This part of the system was given a very high importance since it introduces the communication between well-performing and poorly-performing farmers, as well as the agronomists and farmers. From now on, each development phase may be considered as a new MVP release of the product.

Moreover, the agronomists' daily plan functionalities are delivered with the use of previously introduced *VisitsPlanner*.

In addition, the integration with the external systems is started by implementing *WeatherForecastService*.

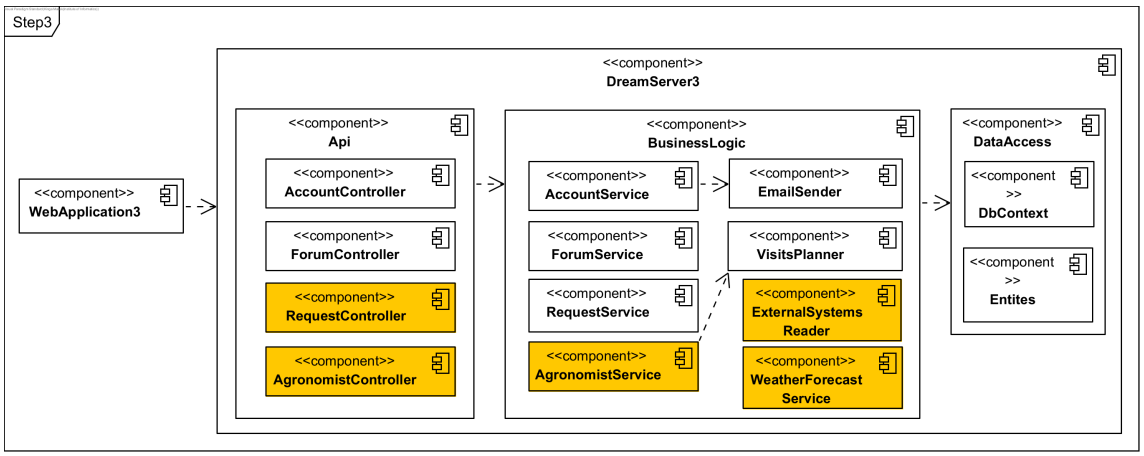


Figure 5.3: Step 3 of the implementation plan. Components added in this step are highlighted with orange color.

Step four concentrates on the integration with the external systems' data by introducing the *WeatherForecastController* as well as the *FarmerController*. The former provides information about short and long-term weather forecasts, and the latter delivers data from the farms' irrigation and sensor systems. The information from the external systems will be available in the database thanks to the *ExternalSystemsReader* built in the previous step (**D6** resolved). The components developed in this phase are presented in the figure 5.4. The implementation

of the *FarmerController* can be done since the dependency **D5** has been resolved in the step 2. Moreover, the farmers may add their production data to the system and agronomists are able to assess the performance of a given farmer.

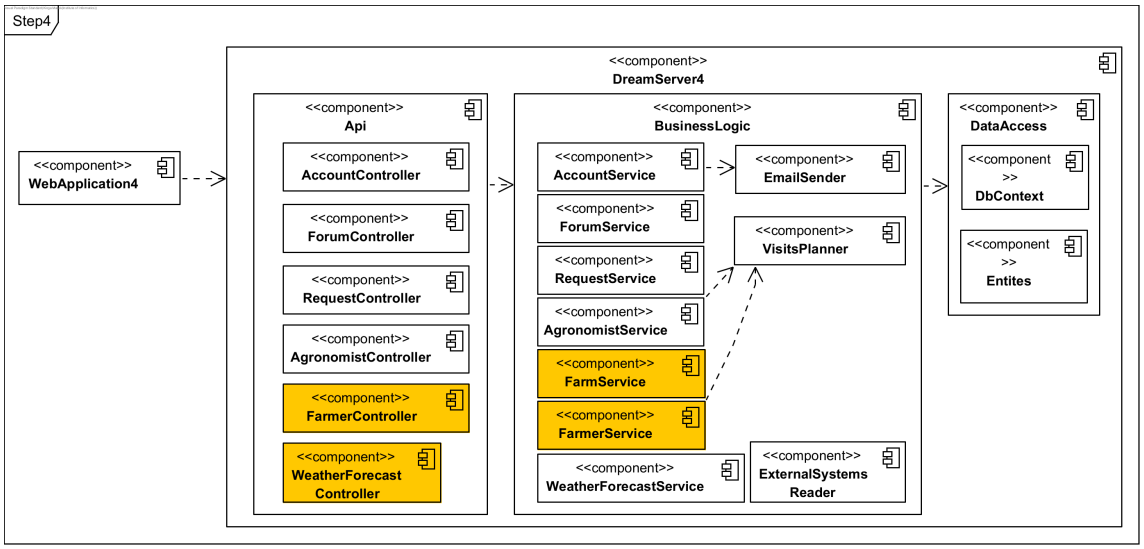


Figure 5.4: Step 4 of the implementation plan. Components added in this step are highlighted with orange color.

In the last development phase, the suggestions' functionality is delivered. From now on, the agronomists are able to manage the suggestions for mandals in their area of responsibility. Besides, the system presents personal suggestions for the farmers in their dashboard screen based on the location of their farm and production type. The last implementation step is presented in the figure 5.5.

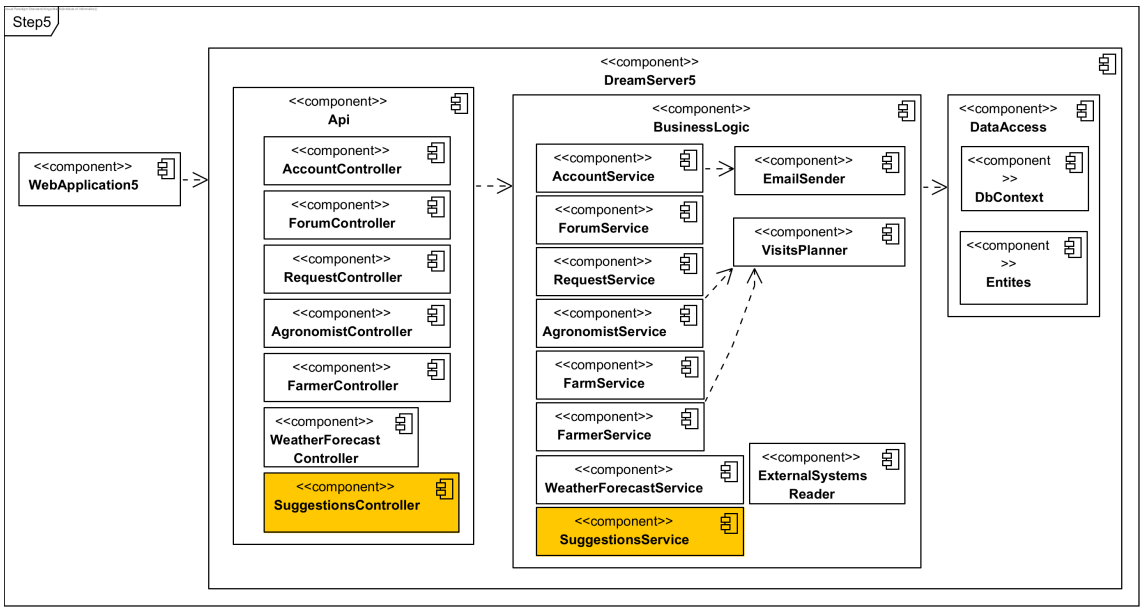


Figure 5.5: Step 5 of the implementation plan. Components added in this step are highlighted with orange color.

## 5.2 Integration and test plan

A comprehensive testing approach must accompany the application's implementation. The code must use abstractions, which makes it possible to use dependency injection and stub individual functionalities and classes, as described in section 2.7.5.

During the development process, individual functions should be tested via means of unit testing. Each finished component requires a set of unit tests, to make sure that it works correctly by itself. A specific emphasis should be placed on components that incorporate custom algorithms, like *VisitsPlanner*.

The system is being released in iterations, as described in the previous section. Each iteration introduces a new group of components, that often need to cooperate to deliver certain functionalities. Those component groups should be tested via means of integration testing to ensure that they work properly together as well as in combination with previously introduced components.

The product released during intermediate phases is a functional system by itself and should be tested as a whole. This is realized via means of system testing. Tests are conducted from the perspective of the end user, and the use cases that are defined in RASD are used to ensure that the scenarios can be implemented and that the system meets the imposed requirements. These tests are performed both manually and automatically, and are executed on the system version most similar to the production build. The released product is also reviewed by the customers itself, who check that the system behaves correctly, applying the use cases as test cases.

During the last phase, a final version of the product is delivered. This version should be tested via the same means as the versions released during intermediate phases.

# Chapter 6

# Effort Spent

## Mariusz Wiśniewski

| Topic | Hours | Date |
|---|---|---|
| Document structure | 1 | 22.12.2021 |
| Introduction | 3 | 22.12.2021 |
| Description of the visit scheduling algorithm | 3 | 25.12.2021 |
| User interface design | 2 | 25.12.2021 |
| Requirements Traceability | 4 | 25.12.2021 |
| Document review | 1 | 02.01.2022 |
| Components mapping | 2 | 03.01.2022 |
| Document verification and refinements | 1 | 04.01.2022 |
| Component interfaces and database model review | 1 | 06.01.2022 |
| Document review and sequence diagrams planning | 2 | 06.01.2022 |
| Sequence diagrams | 2 | 06.01.2022 |
| Runtime view and sequence diagrams fixes | 2 | 06.01.2022 |
| Integration and test plan discussion and adjustments | 1 | 09.01.2022 |
| Total | 25 | |

## Józef Piechaczek

| Topic | Hours | Date |
|---|---|---|
| Deployment diagram | 2 | 29.12.2021 |
| Deployment overview | 1,5 | 31.12.2021 |
| Document review | 1 | 02.01.2022 |

| | | |
|---|---|---|
| Requirements Traceability, Additional Mocks | 4 | 03.01.2022 |
| High level components | 2,5 | 05.01.2022 |
| Mocks description | 2 | 05.01.2022 |
| Document review and sequence diagrams planning | 2 | 06.01.2022 |
| Mocks description | 2 | 06.01.2022 |
| Mocks description | 3 | 07.01.2022 |
| Testing phase | 3 | 08.01.2022 |
| Total | 23 | |

## Kinga Marek

| Topic | Hours | Date |
|---|---|---|
| Review of the visit scheduling algorithm | 1 | 26.12.2021 |
| High-level diagram and architecture overview | 2,5 | 28.12.2021 |
| Dream server component diagram | 3 | 28.12.2021 |
| Selected Architectural Styles and Patterns (REST, N-tier, DI) | 1,5 | 28.12.2021 |
| Dream server components diagram description | 1 | 31.12.2021 |
| Document review | 1 | 02.01.2022 |
| Database model | 2 | 3.01.2022 |
| Component interfaces | 1 | 3.01.2022 |
| Component interfaces | 1,5 | 4.01.2022 |
| Component diagram extension | 0,5 | 4.01.2022 |
| Component interfaces and database model description | 1,5 | 5.01.2022 |
| Document review and sequence diagrams planning | 2 | 06.01.2022 |
| Functional requirements mapping improvements (section 4.2) | 1 | 06.01.2022 |
| Implementation plan | 5,5 | 07.01.2022 |
| Integration and test plan discussion and adjustments | 1 | 09.01.2022 |
| Total | 26 | |

# References


[3] Auth0. *JWT*. https://jwt.io/. (Visited on 01/05/2022).

[4] Microsoft Corporation. *Common web application architectures*. https://docs.microsoft.com/en-gb/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures. (Visited on 12/29/2021).

[5] Microsoft Corporation. *Dependency injection in ASP.NET Core*. https://docs.microsoft.com/en-gb/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-6.0. (Visited on 12/29/2021).

[6] Microsoft Corporation. *RESTful web API design*. https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design. (Visited on 12/29/2021).

[7] Microsoft Corporation. *Task-based asynchronous pattern*. https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap?redirectedfrom=MSDN. (Visited on 01/05/2022).

[8] Inheritance Entity Framework Core. *Task-based asynchronous pattern*. https://docs.microsoft.com/en-us/ef/core/modeling/inheritance. (Visited on 01/05/2022).

[9] EntityFrameworkTutorial.net. *DbContext in Entity Framework*. https://www.entityframeworktutorial.net/entityframework6/dbcontext.aspx. (Visited on 01/05/2022).

[10] Facebook. *Creating a Production Build*. https://create-react-app.dev/docs/production-build. (Visited on 01/07/2022).

[11] Roy Thomas Fielding. "REST: Architectural Styles and the Design of Network-based Software Architectures". Doctoral dissertation. University of California, Irvine, 2000. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[12] The PostgreSQL Global Development Group. *PostgreSQL*. https://www.postgresql.org. (Visited on 01/04/2022).

[13] Jerzy Kisielnicki and Anna Maria Misiak. "Effectiveness of agile compared to waterfall implementation methods in it projects: Analysis based on business intelligence projects". In: *Foundations of Management* 9.1 (2017), pp. 273–286.

[14] Microsoft. *Common Language Runtime (CLR) overview*. https://create-react-app.dev/docs/production-build. (Visited on 01/07/2022).

[15] Microsoft. *Kestrel web server implementation in ASP.NET Core*. https://create-react-app.dev/docs/production-build. (Visited on 01/07/2022).

[16] Will Reese. "Nginx: The High-Performance Web Server and Reverse Proxy". In: *Linux J.* 2008.173 (Sept. 2008). ISSN: 1075-3583.

[17] Günther Schuh, Christian Dölle, Sebastian Schloesser, et al. “Agile Prototyping for technical systems–Towards an adaption of the Minimum Viable Product principle”. In: *DS 91: Proceedings of NordDesign 2018, Linköping, Sweden, 14th-17th August 2018* (2018).

[18] H. Frystyk T. Berners-Lee R. Fielding. *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945. RFC Editor, May 1996. URL: https://www.rfc-editor.org/rfc/rfc1945#section-5.1.1.

[19] Wikipedia contributors. *Object–relational mapping — Wikipedia, The Free Encyclopedia*. [Online; accessed 5-January-2022]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Object%E2%80%93relational_mapping&oldid=1047542743.