

POLITECNICO
MILANO 1863

Computer Science and Engineering

Software Engineering 2

Academic year 2021-2022

Design Document



Data-dRiven PrEdictive FArMing in Telangana

Authors:

| | |
|------------------|--------|
| Arslan ALI | 971503 |
| Elisa SERVIDIO | 996387 |
| Federica SURIANO | 953085 |

January 08, 2022

Version 1.0

Contents

| | |
|--|-----------|
| List of Tables | 3 |
| List of Figures | 4 |
| 1 Introduction | 5 |
| 1.1 Purpose | 5 |
| 1.2 Scope | 5 |
| 1.3 Definitions, Acronyms, Abbreviations | 5 |
| 1.3.1 Definitions | 6 |
| 1.3.2 Acronyms | 8 |
| 1.3.3 Abbreviations | 8 |
| 1.4 Revision History | 9 |
| 1.5 Reference Documents | 9 |
| 1.6 Document Structure | 9 |
| 2 Architectural Design | 11 |
| 2.1 Overview | 11 |
| 2.1.1 Class Diagram | 13 |
| 2.2 Component View | 15 |
| 2.3 Deployment View | 20 |
| 2.4 Runtime View | 23 |
| 2.4.1 Farmer registers | 23 |
| 2.4.2 Farmer logs in | 25 |
| 2.4.3 Farmer inserts data about his/her production | 27 |
| 2.4.4 DREAM sends suggestions to farmer | 29 |
| 2.4.5 Farmer searches for a thread in the discussion forum | 31 |
| 2.4.6 Farmer creates a new thread in the discussion forum | 33 |
| 2.4.7 Farmer replies in the discussion forum | 35 |
| 2.4.8 Farmer creates a help request | 37 |
| 2.4.9 Agronomist replies to a help request | 39 |
| 2.4.10 Farmer solves a help request | 41 |
| 2.4.11 Agronomist visualizes weather conditions | 43 |
| 2.4.12 Agronomist visualizes and analyzes farmers' data | 45 |

| | | |
|----------|--|-----------|
| 2.4.13 | Agronomist updates his/her daily plan | 47 |
| 2.4.14 | Agronomist confirms his/her daily plan | 49 |
| 2.4.15 | Policy Maker visualizes and analyzes farmers' data | 51 |
| 2.5 | Component Interfaces | 53 |
| 2.6 | Selected Architectural Style and Patterns | 55 |
| 2.6.1 | Architectural Style: Client-Server | 55 |
| 2.6.2 | Architectural Pattern: Model-View-Controller Structure | 56 |
| 2.7 | Other Design Decisions | 56 |
| 2.7.1 | ID code for policy maker and agronomist from external database | 57 |
| 2.7.2 | Internal database replication | 57 |
| 3 | User Interface Design | 58 |
| | Farmer | 58 |
| | Agronomist | 59 |
| 4 | Requirement Traceability | 60 |
| 5 | Implementation, Integration and Test Plan | 67 |
| 5.1 | Implementation Plan | 67 |
| 5.2 | Integration Strategy | 69 |
| 5.3 | Testing Plan | 73 |
| 6 | Effort Spent | 74 |
| 6.0.1 | Ali Arslan | 74 |
| 6.0.2 | Servidio Elisa | 75 |
| 6.0.3 | Suriano Federica | 75 |

List of Tables

| | | |
|-----|---|----|
| 1.1 | Definitions | 6 |
| 1.2 | Acronyms | 8 |
| 1.3 | Abbreviations | 8 |
| 1.4 | Revision History | 9 |
| 6.1 | Effort spent - Ali Arslan | 74 |
| 6.2 | Effort spent - Servidio Elisa | 75 |
| 6.3 | Effort spent - Suriano Federica | 75 |

List of Figures

| | | |
|------|--|----|
| 2.1 | System Architecture | 12 |
| 2.2 | Class Diagram | 14 |
| 2.3 | Component legend | 15 |
| 2.4 | Component diagram | 19 |
| 2.5 | Deployment diagram | 22 |
| 2.6 | Farmer registers | 24 |
| 2.7 | Farmer logs in | 26 |
| 2.8 | Farmer inserts data about his/her production | 28 |
| 2.9 | DREAM sends suggestions to farmer | 30 |
| 2.10 | Farmer searches for a thread in the discussion forum | 32 |
| 2.11 | Farmer creates a new thread in the discussion forum | 34 |
| 2.12 | Farmer replies on the discussion forum | 36 |
| 2.13 | Farmer creates a help request | 38 |
| 2.14 | Agronomist replies to a help request | 40 |
| 2.15 | Farmer solves a help request | 42 |
| 2.16 | Agronomist visualizes weather conditions | 44 |
| 2.17 | Agronomist visualizing and analyzing farmers' data | 46 |
| 2.18 | Agronomist updates his/her daily plan | 48 |
| 2.19 | Agronomist confirms his/her daily plan | 50 |
| 2.20 | Policy Maker visualizes and analyzes farmers' data | 52 |
| 2.21 | Component Interfaces diagram | 54 |
| 2.22 | Model-View-Controller structure | 56 |
| 3.1 | Farmer Interfaces | 58 |
| 3.2 | Agronomist Interfaces | 59 |
| 5.1 | Integration of Database and DatabaseAccess components. | 69 |
| 5.2 | Login and registration services integration. | 70 |
| 5.3 | Integration of components for the three actors | 71 |
| 5.4 | Integration of all main components | 72 |

Chapter 1

Introduction

1.1 Purpose

The goal of DD (Design Document) is to provide a more extensive overview of the architectural decisions and their communication interfaces made to implement all the functionalities described in the RASD document for what concerns the DREAM system. In *Chapter 5* it will also present implementation, integration and test plan.

1.2 Scope

As stated in the RASD document, the aim of the DREAM software product is to develop and adopt anticipatory governance models for food systems to strengthen data-driven state policy. It takes care of the acquisition and management of all data collected in order to support the work of farmers, agronomists and policy makers. The system aims to collect data not only from sensors located throughout the territory, but also from farmers. The analysis of the acquired data aims to improve the production of farmers. Bad performing farmers are helped by their related agronomist and by well performing farmers. Everything is supervised by agronomists who take care of their own geographical area of competence.

1.3 Definitions, Acronyms, Abbreviations

In the following section is clarified the meaning of some definitions, acronyms and abbreviations which will be used in the DD, in order to help the general understanding of the document.

1.3.1 Definitions

Table 1.1: Definitions

| | |
|-------------------------------|--|
| <i>The system</i> | The whole system to be developed |
| <i>User</i> | A farmer/agronomist/policy maker who uses the application |
| <i>Unregistered user</i> | A farmer/agronomist/policy maker who is not yet registered into the application |
| <i>Application service</i> | Functionality offered by the system for certain users |
| <i>ID Code</i> | Unique code that identifies each agronomist and policy maker |
| <i>Policy maker</i> | The user of the application who decides about new policies for Telangana |
| <i>Farmer</i> | The user of the application who owns or manages a farm |
| <i>Address</i> | The address inserted by the farmer which corresponds to his/her farm's location. It is composed by city, zip code, street and number |
| <i>Performance</i> | Indicator of the progress of a farmer's activity up to a certain date. Its value is calculated as numerical score |
| <i>Score</i> | Performance rating computed by a function that depends on: type and quantity of harvested product, weather conditions, quantity of water consumed, soil moisture |
| <i>Well performing farmer</i> | A farmer who has a score higher than a certain threshold |
| <i>Bad performing farmer</i> | A farmer who has a score below a certain threshold |

| | |
|-------------------------|---|
| <i>Agronomist</i> | The user of the application dealing with the management of a certain mandal |
| <i>Mandal</i> | A local government area and administrative division. Telangana is subdivided into districts which are themselves subdivided into mandals |
| <i>Daily plan</i> | Application service that allows the agronomist to manage his/her daily work schedule. Specifically, it allows him/her to track and organize visits to farmers |
| <i>Discussion forum</i> | Application service which a farmer can use to exchange ideas and opinions about a topic |
| <i>Post</i> | Contributions of the participants to the discussion placed one below the other in sequence |
| <i>Thread</i> | It is composed of the topic followed by the posts left by the various participants in the discussion |
| <i>Help request</i> | Application service which a farmer can use to request for help to the agronomist or/and other well performing farmers |
| <i>Help response</i> | Reply to a help request |
| <i>Tier</i> | A machine or a group of machines |
| <i>Layer</i> | Separate functional component that interact with other layers in some sequential and hierarchical way |
| <i>Mockup</i> | Full-size model of a design or device, used for product presentations or other purposes |

1.3.2 Acronyms

Table 1.2: Acronyms

| | |
|--------------|---|
| <i>DREAM</i> | Data-dRiven PrEdictive FArMing in Telangana |
| <i>RASD</i> | Requirement Analysis and Specification Document |
| <i>DD</i> | Design Document |
| <i>API</i> | Application Programming Interface |
| <i>DBMS</i> | Database Management System |
| <i>DB</i> | Database |
| <i>DMZ</i> | Demilitarized zone |
| <i>IN</i> | Internal Network |
| <i>MVC</i> | Model-view-controller |
| <i>UI</i> | User Interface |
| <i>HTTP</i> | Hypertext Transfer Protocol |

1.3.3 Abbreviations

Table 1.3: Abbreviations

| | |
|------------|------------------|
| <i>R.i</i> | i-th requirement |
|------------|------------------|

1.4 Revision History

Table 1.4: Revision History

| Version | Date | Authors | Summary |
|---------|------------|--|---------------|
| 1.0 | 08/01/2022 | Arslan Ali Elisa Servidio Federica Suriano | First release |

1.5 Reference Documents

- Specification document: “R&DD Assignment A.Y. 2021/2022”
- Software Engineering 2 course slides A.Y. 2021/2022 - Politecnico di Milano
- UML official specification <https://www.omg.org/spec/UML/>
- *Enoch, Simon & Ge, Mengmeng & Hong, Jin & Alzaid, Hani & Kim, Dan. (2018). A Systematic Evaluation of Cybersecurity Metrics for Dynamic Networks. Computer Networks.*
- *Abdullah, Muhammad & Iqbal, Waheed & Bukhari, Faisal. (2019). Containers vs Virtual Machines for Auto-scaling Multi-tier Applications Under Dynamically Increasing Workloads.*

1.6 Document Structure

The DD is structured in the following six chapters:

- **Chapter 1 - Introduction:** Brief description of the document that will be presented, it gives a general information about what this document is going to explain.
- **Chapter 2 - Architectural Design:** This chapter describes the architectural choices adopted to build DREAM, with an overview of the main components and their interactions, in the final section it also describes some deployment choices.
- **Chapter 3 - User Interface Design:** Mockups of Mobile UIs of system.

- **Chapter 4 - *Requirement Traceability*:** Chapter that explains how the requirements listed in the RASD maps design choices made in this document.
- **Chapter 5 - *Implementation, Integration and Test Plan*:** The last chapter describes some guidelines that will be used to implement DREAM, in which order components and sub-components are integrated and how they are tested.
- **Chapter 6 - *Effort Spent*:** Effort spent by team members, a table that contains the hours required to realize this document.

Chapter 2

Architectural Design

In this chapter are reported the main architectural design choices from a high level point of view. In the following sections are presented the chosen paradigm of the system and its components, followed by the description of how the system will be deployed. Moreover, components' sequence and interface diagrams are listed to describe the way components interact to accomplish specific tasks of the application. Eventually, the last two sections provide specifications regarding the selected architectural style and other design choices.

2.1 Overview

The software to be developed is a distributed application with a client-server multi-tiered architecture.

The general architecture of the system can be subdivided into the three logic layers:

- *Presentation Layer*: it handles the interaction with all kinds of user. It provides an interface to users accessing both the mobile application and the web application allowing them to exploit DREAM's services in the most efficient and intuitive way.
- *Application Layer*: it contains business and data access logic. It handles the functions to be provided for the users. It also manages communication between the end user interface and the database.
- *Data Layer*: it manages the persistent storage and retrieval of data accessing to the database.

The three logic software layers are built on top of a three tier architecture, which consists in three different hardware tiers that represent a machine (or a group of machines):

- *Client tier*: it consists of the client programs that are used to exploit system's services and the devices where those programs are installed, which in this case are pc for the web application and smartphone for the mobile applications.
- *Logic tier*: it controls the application's functionality by performing detailed processing. It dynamically generates content in various formats for the client tier from which it collects inputs and return appropriate results from the components in the logic tier. It also guarantees the interactions between the Client and the Data tier. It contains Web and Application Servers.
- *Data tier*: runs the DBMS and holds all of the sensitive application data. It contains database servers.

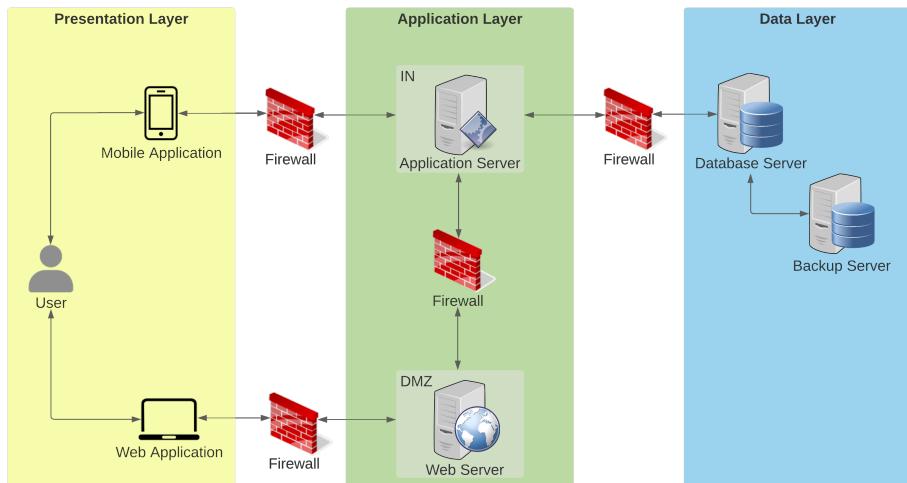


Figure 2.1: System Architecture

The Application Server is the core of the system's architecture as it is the central point between all the other components. It holds the business logic of the system with Web Server. Web and Application servers communicate to each other. It has persistent connection with Data tier and by communicating with its DBMS it can handle all the operations of reading, writing and uploading of data.

The service is supposed to be accessed through both a web application and a mobile application, and this is valid for all kinds of users. Users who

access DREAM through web application can communicate with the Web Server while users logged into the mobile application communicate directly with the Application Server.

To guarantee high level of security due to the sensitivity of treated data, firewalls are installed.

Specifically, there are firewalls between the Internet accessed by the customer to use the application and the Web Server and between the latter and the Application Server in order to create a demilitarized zone (DMZ), so that the external network can access only to the resources exposed in the DMZ. In order to ensure secure access to the database and to delimit the internal network (IN), a firewall is placed between the Application Server and the Database Server.

Moreover, the Database Server periodically stores critical data in the Backup Server.

The described architecture has been chosen because tiering ensures the ability to scale applications, guaranteeing the system scalability and flexibility to future integrations and modifications. The servers that constitute each tier in fact may differ both in capability and number. If needed, it can be possible to add more servers (horizontal scaling or scaling out) or more powerful servers (vertical scaling or scaling up) to each singular tier.

2.1.1 Class Diagram

In the RASD document a Class Diagram has been showed to model the problem, including all the classes involved without going into too much detail. For this reason, in this section an updated Class Diagram is introduced to represent the system from an implementation oriented point of view.

The following diagram differs from the one presented in the RASD in two aspects:

- the main methods of each kind of user are presented;
- attributes data types are specified.

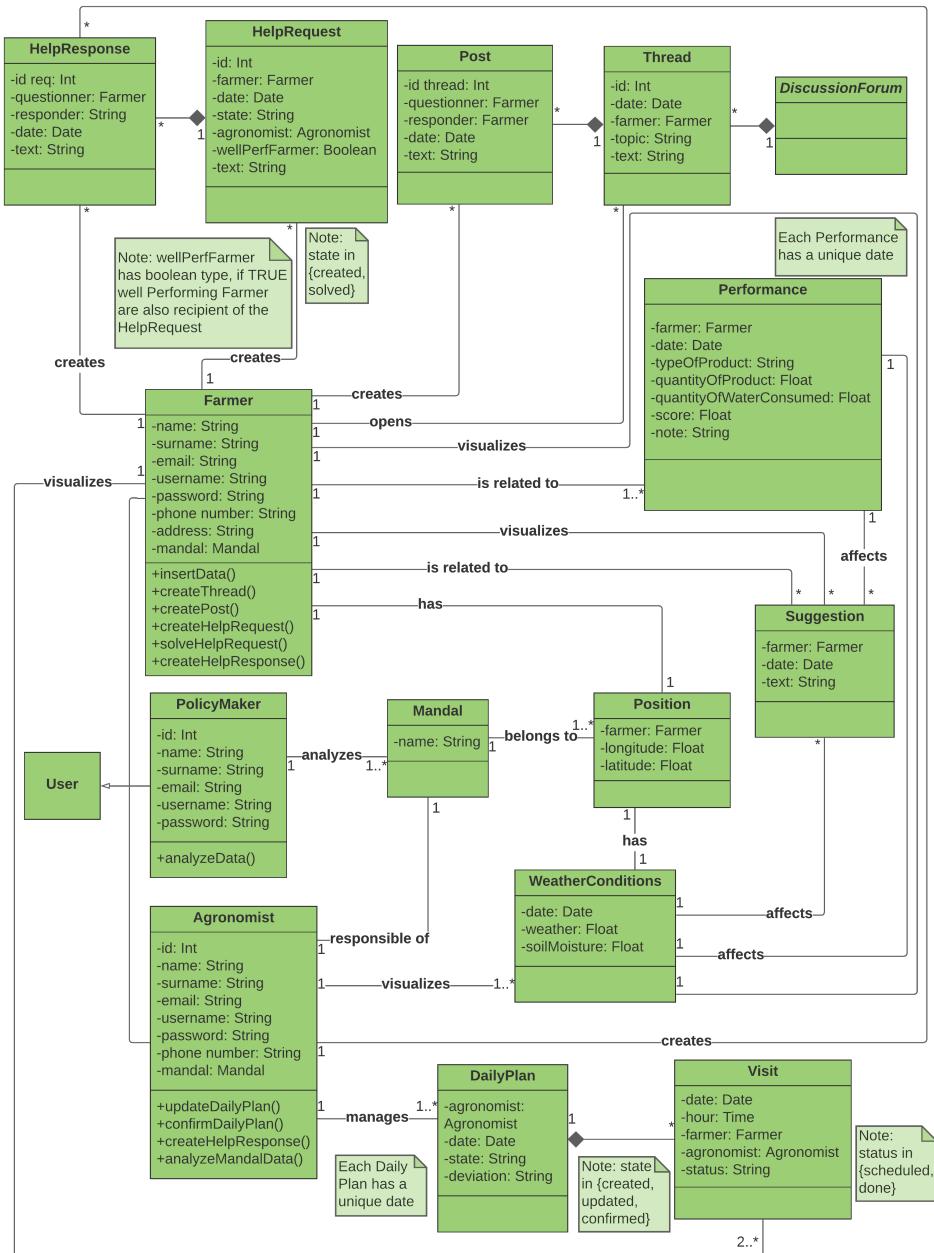


Figure 2.2: Class Diagram

2.2 Component View

The purpose of this section is to show the component diagram which is intended to represent the internal structure of the modeled software system in terms of its main components and the relationships among them.

The following legend is used:

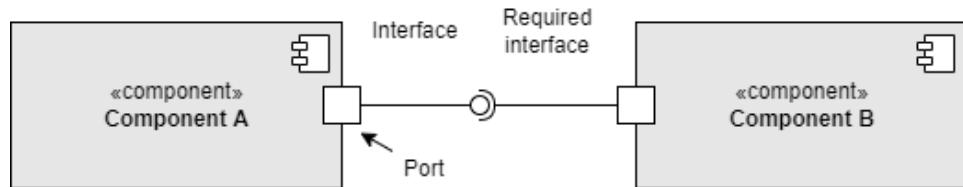


Figure 2.3: Component legend

All components within the diagram are described in detail below.

- **WebApplication:** it manages all user interactions with the web application and associates them with the required functions. It deals with managing all notifications addressed to a specific user.
- **MobileApplication:** it manages all user interactions with the mobile application and associates them with the required functions. It deals with managing all notifications addressed to a specific user.
- **AuthenticationService:** it is a service that is involved when login is performed (both from the mobile app and the web app). It checks user permissions according to the role she/he has. In order to verify the credentials inserted by the user, this component is directly connected with DatabaseAccess component.
- **RegistrationService:** it manages the registration of a unregistered user; in particular, it allows to retrieve farmer's position thanks to the connection to GoogleMapsAPI component. It also accesses the database managed by the Telangana government to check the validity of the ID code entered by the policy maker and agronomist during registration in DREAM. Access to this database takes place thanks to the TelanganaGovernmentService component. In order to store the new user in the database, the component must be connected to the DatabaseAccess component. RegistrationService component includes the following subcomponent.

- **EmailManager:** it is necessary when a user who has not yet registered wants to register. To verify his/her email during registration, the system automatically sends a confirmation email to the user and activates a 10-minute timer within which the user must confirm his/her registration.
- **PolicyMakerService:** This component includes the following sub-components.
 - **TimeChartService:** it computes and displays Telangana's farmers data in a graph so that the policy maker can analyze the various parameters of interest over time. In order to collect all the data of interest, the component must be connected to the DatabaseAccess component.
 - **MapService:** it computes and displays Telangana's farmers performance map using some filters. In order to collect all the data of interest, the component must be connected to the DatabaseAccess component. The connection to GoogleMapsAPI component allows to process the map to be displayed.
- **AgronomistService:** This component includes the following subcomponents.
 - **HelpResponseService:** it allows an agronomist to manage the response to a help request received. In order to retrieve all the help requests with a certain agronomist as a recipient and store the new help response created by him/her, the component must be connected to the DatabaseAccess component.
 - **MandalMapService:** it retrieves and displays relevant personal data of farmers and their performance score on the map of a mandal. This component is closely linked to the TableService component, which is called up as soon as the agronomist selects a farmer of interest on the map. Obviously, in order to collect all the data of interest, the component must be connected to the DatabaseAccess component. The connection to GoogleMapsAPI component allows to process the map to be displayed.
 - **TableService:** it retrieves and displays the table of farmers' performance in descending order based on their performance score. Obviously, in order to collect all the data of interest, the component must be connected to the DatabaseAccess component.
 - **DailyPlanService:** it deals with the creation and updating and confirmation of the daily plan of a agronomist. Specifically, it guarantees that for each farmer there are at least two visits per

year by the responsible agronomist. To retrieve the scheduled plan, this component must be connected with DatabaseAccess component.

- **WeatherForecastsService:** it collects the data related to the weather forecasts in the mandal of a certain agronomist according to the date selected by the agronomist. To retrieve weather forecasts, it needs to be connected to the external API TelanganaGovernmentService component. The connection to CopernicusClimateDataStoreService component allows to retrieve soil moisture in the mandal. These data are placed on the appropriate mandal map thanks to an external API component: GoogleMapsAPI.
- **FarmerService:** This component includes the following subcomponents.
 - **HelpRequestService:** it is necessary for the creation and subsequent management of help requests by the farmer. The connection to DatabaseAccess component allows it to retrieve the recipients to whom the request can be sent (agronomist in charge of the mandal and well performing farmers).
In addition, if a (well performing) farmer has received a help request from another farmer, this component takes care of managing the function that allows the farmer to respond to the request.
 - **DiscussionForumService:** it manages the creation of threads by a farmer on the discussion forum, the creation of posts in response to a specific thread (always by other farmers) and the search by topic of a thread already created. It also performs these functions thanks to the connection to the DatabaseAccess component.
 - **VisitService:** it retrieves the visits scheduled for a farmer that corresponds to the daily plan of an agronomist. Since visits are stored in the database, to retrieve them, this component must be connected to the DatabaseAccess component.
 - **PerformanceService:** it takes care of updating a farmer's performance data every time he/she enters new data about his/her crop. It is connected to the DatabaseAccess component not only to store the computed performance but also to get farmer's position and his/her last performance date (to take into account the right time interval for which performance has to be computed). It also is connected to external API TelanganaWaterIrrigationService to compute the quantity of water consumed. Furthermore it is linked to WeatherConditionService component which allows to retrieve weather conditions and soil moisture.

- **WeatherConditionService:** it collects the data related to weather conditions based on the farmer's position. To accomplish this task, it needs to be connected to the external API TelanganaGovernmentService as well as to the CopernicusClimateDataStoreService component. It also has to be connected to DatabaseAccess component to retrieve farmer's position.
- **SuggestionsService:** it is used to elaborate farmer's suggestions based on his/her relevant information regarding his/her production and the position of his/her farm. These data are obtained thanks to the connection to the DatabaseAccess and WeatherConditionService components.
- **DatabaseAccess:** it is responsible to retrieve (and store) data from (to) a data source. In this case, it manages the access to the database.
- **Database:** This component includes the following subcomponent.
 - **DBMS:** Database Management System refers to a software that allow users to access databases and manipulate, maintain, report, and relate data. It is used to reduce data redundancy, share data in a controlled manner, and reduce data integrity issues.
- **ExternalAPIs:** This component includes the following subcomponents.
 - **GoogleMapsAPI:** it allows the connection of the application to Google Maps APIs. All functions that require access to geographical data, not yet saved in the internal database, will rely on this component.
 - **CopernicusClimateDataStoreService:** it is responsible for retrieving soil moisture data in the Telangana region.
 - **TelanganaGovernmentService:** it takes care of retrieving data relating to weather conditions in Telangana region. It also used to access ID codes' DB managed by Telangana government to check validity of ID code inserted by policy makers and agronomist while registering into DREAM.
 - **TelanganaWaterIrrigationService:** it is responsible for retrieving the amount of water consumed by each farmer in the Telangana region.

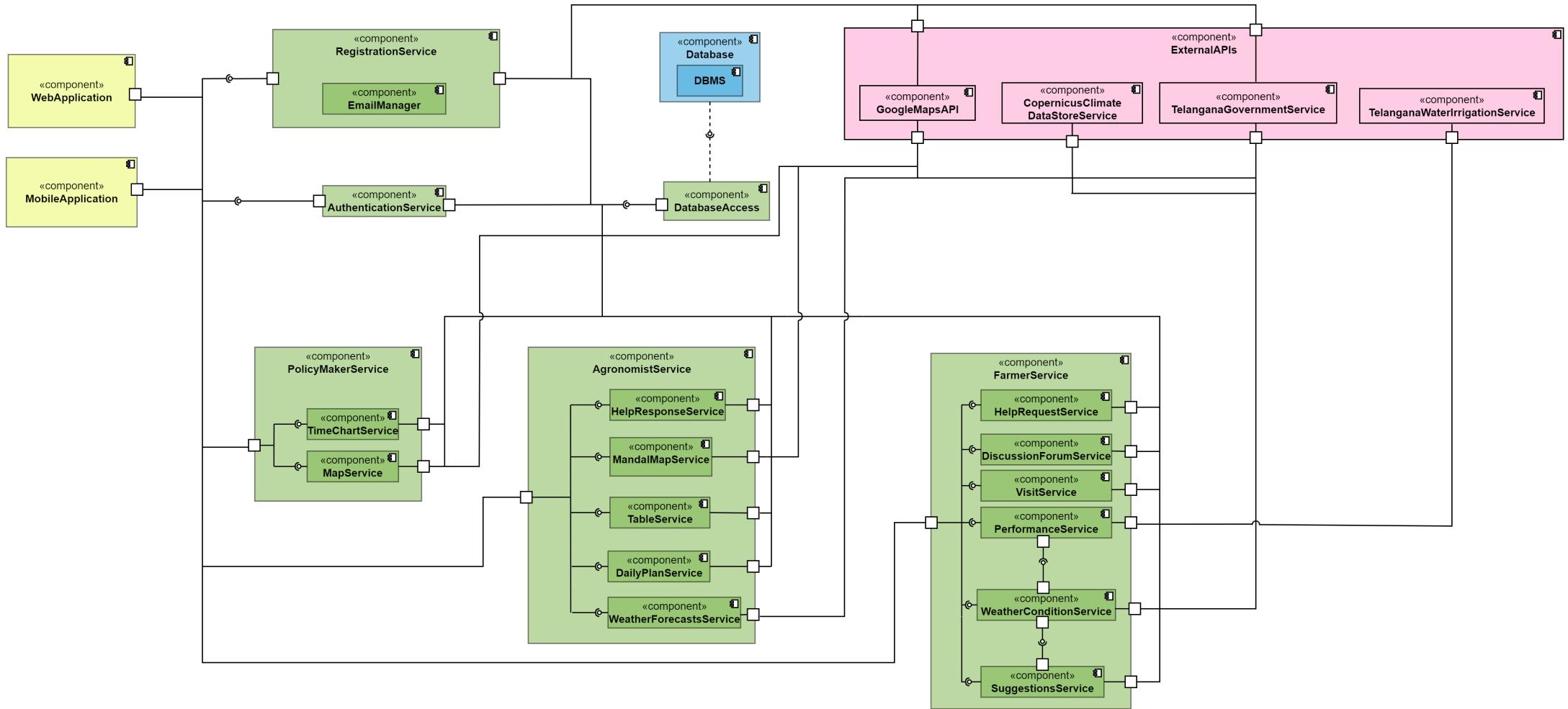


Figure 2.4: Component diagram

2.3 Deployment View

This section shows the physical distribution of the system, that is, the environment in which the system runs. Specifically, *Figure 2.5* shows the structure of the hardware components that execute the software.

As previously mentioned, the system is divided into three tiers, whose components are described below:

- **Client Tier:** it consists of the client programs used to exploit system services and the devices on which these programs are installed.
 - **Smartphone:** users access the DREAM system and use the related services through a mobile application from a smartphone.
 - **Personal Computer:** users access the DREAM system and use the related services through the Web browser from a personal computer.
- **Logic Tier:** it consists of components that handle the interaction between client tier and the data tier. It contains Web and Application Servers.
 - **Web Server:** it provides content to the Web using the HTTP(S) protocol. It mainly works to serve the requested web pages continuously and without interruption. As long as the Web Server is up and running, the corresponding web pages and sites will be available to users on the network.
 - **Application Server:** it hosts and exposes business logic and processes. It provides the infrastructure and logical capabilities to support, develop and run applications in a distributed context. The Application Server communicates directly with the Mobile Application through HTTP(S) protocol, in order to create a lightweight API to request only the necessary data. The Application Server will receive this call, make the call to the external service, filter it, and return only what the application has requested. This will reduce the amount of data transiting over mobile Internet and thus increase application performance.
- **Data Tier:** it runs the DBMS and holds all of the sensitive application data. It contains database servers.
 - **Database Server:** it allows access to one or more Database Systems and is an essential support for Web Servers in managing the delivery and storage of data. It communicates with the Application Server via the standard TCP/IP protocol and provides access to the physical database which is in charge of storing all persistent data of users who use DREAM.

Firewalls are also included in the system architecture to delimit safe areas within the system. Specifically, there are firewalls between the Internet accessed by the customer to use the application and the Web Server and between the latter and the Application Server in order to create a demilitarized zone (DMZ). The purpose of the DMZ is to add an additional level of security to an internal network, where a node belonging to an external network can only access the services made available, without putting at risk and compromising the security of the entire network. In order to ensure secure access to the database and to delimit the internal network (IN), a firewall is placed between the Application Server and the Database Server.

In order to distribute the load of the requests to the DREAM website among several servers, **load balancers** are introduced for the Web Server. In this way, the reliability and scalability of the entire architecture increases because requests will be shared equally among the different servers. If one of the servers is unable to respond to the request, the user can still use the service based on one of the active servers. Similarly, the Web Server may need to query the database to generate responses. Web Servers send database queries to the load balancer which balances the inbound database workload on the database servers.

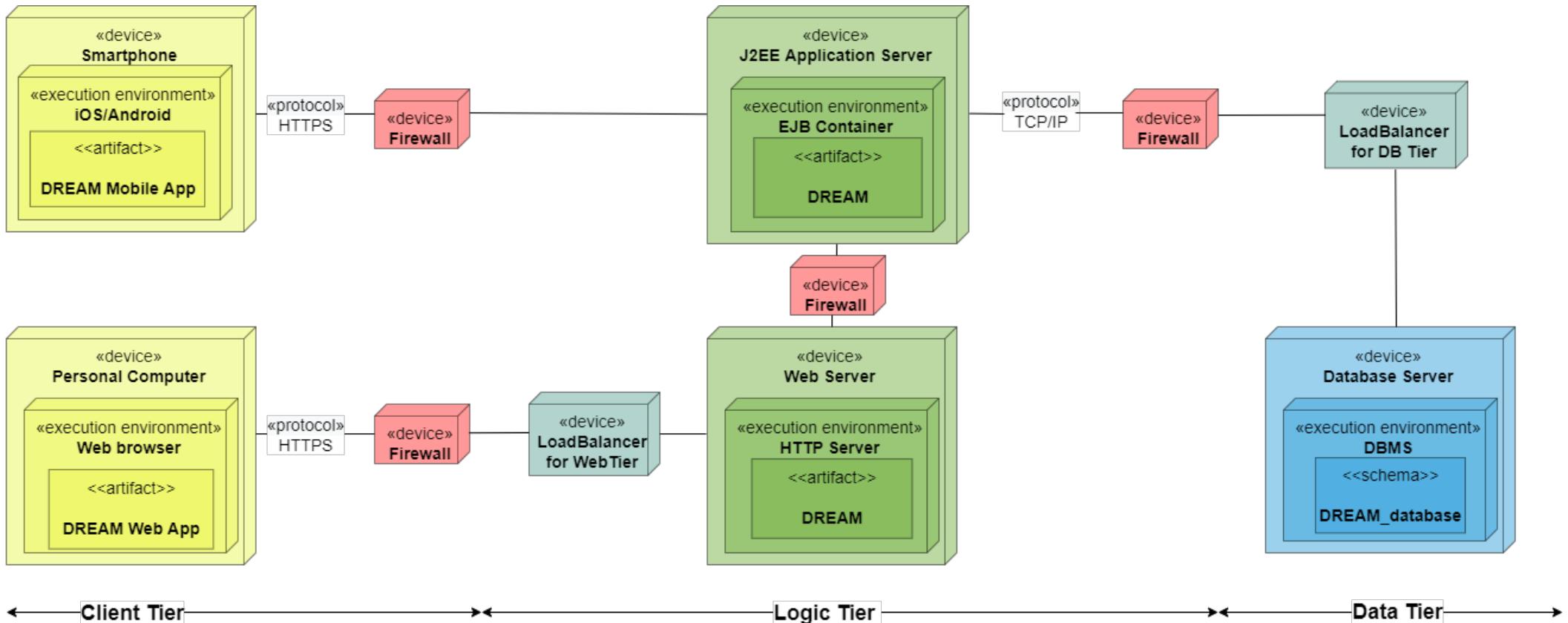


Figure 2.5: Deployment diagram

2.4 Runtime View

To accomplish specific tasks of DREAM the components listed in *Section 2.2* must interact with each other. To describe the type of interactions which occur between one another, in this section are presented the most relevant sequence diagrams.

All functionalities described by the following diagrams can be performed through both Web and Mobile application by the users.

2.4.1 Farmer registers

In this sequence diagram it is shown the process of registration attended by the farmer. It is supposed that the farmer has already selected his role in the registration form and the Web/Mobile Application shows him/her the related fields.

After having switched from the *login* page to the *registration* one, he/she signs up filling in the registration form. Then, Web/MobileApplication checks that all the credentials inserted by the farmer are valid. Then, the registration request is sent from Web/MobileApplication to RegistrationService component which is in charge of retrieving position and mandal of the farmer (through GoogleMapsAPI) before storing his/her data persistently in the Database. Eventually, DatabaseAccess component is called to store the information in the Database. The result of the operation is propagated back to Web/Mobile Application, following in reverse order the same path as before. If the result is negative the registration should be repeated and an alert regarding registration failure is sent to the farmer. If the result is positive the farmer is redirected to a waiting page and the Web/mobile application calls RegistrationService component to send a confirmation email to the farmer. The function is propagated to EmailManager component which is in charge to check if the farmer has clicked on the confirmation link in the email within a certain time limit. The result of the operation is propagated back to Web/MobileApplication, following in reverse order the same path as before. If the result is negative an alert regarding confirmation failure is sent to the farmer and he/her has to request for the generation of a new confirmation email to repeat the process. If the result is positive the farmer is redirected to *login* page.

The process of registration is different for the agronomist and the policy maker. In fact, the credentials to be inserted differ in particular in a unique id code which is checked calling an appropriate function to be propagated from Web/MobileApplication to RegistrationService. This component is in charge of verifying the id code calling TelanganaGovernmentService component. Eventually, the result of the operation is returned following in reverse order the same path as before and if it is false the filling operation must be repeated.

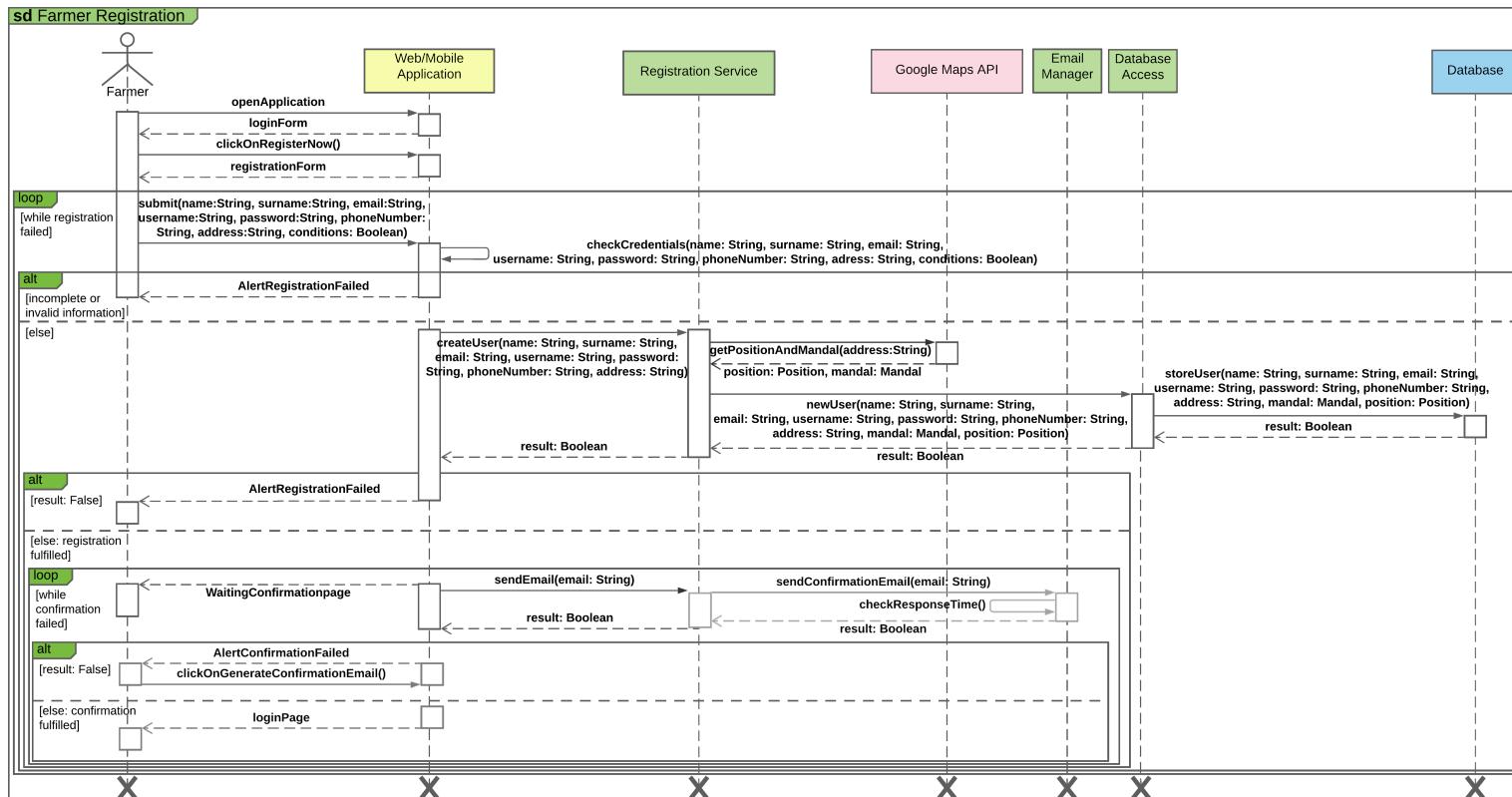


Figure 2.6: Farmer registers

2.4.2 Farmer logs in

This sequence diagram shows the login procedure executed by a farmer who is already registered.

After having opened DREAM application on his device, the farmer fills in the form with his username and password. The inserted credentials are checked by Web/MobileApplication, which controls whether one or both these two fields are missing. If it is the case, it shows to the farmer an alert regarding the login failure. If credentials are not missing, the AuthenticationService is called to check the validity of credentials and this component propagates the request to the DatabaseAccess that provides access to the Database, where all the users credentials are stored persistently. If a couple of username and password that matches the one inserted by the farmer is found, the login is successfully completed, otherwise an alert is shown and the operation must be repeated. This process is the same for all kind of users.

For the sake of completeness, in the sequence diagram it is shown also what is returned to the farmer who has successfully logged in.

The farmer must be able to visualize his/her *homepage*, in which are reported weather conditions regarding his/her position and his/her next visit scheduled in the daily plan. For this reason a request to get weather conditions is propagated from Web/MobileApplication to WeatherConditionService which is able to retrieve weather and soil moisture from external APIs given the position of the user. The result of the operation is propagated to Web/MobileApplication. Then, the latter component send a request to VisitService. The request is propagated to DatabaseAccess to access the Database and retrieve all visits related to the farmer, which are returned eventually to VisitService, component that gets the next visit among them and returns it to Web/MobileApplication. Eventually, the *homepage* is shown to the farmer.

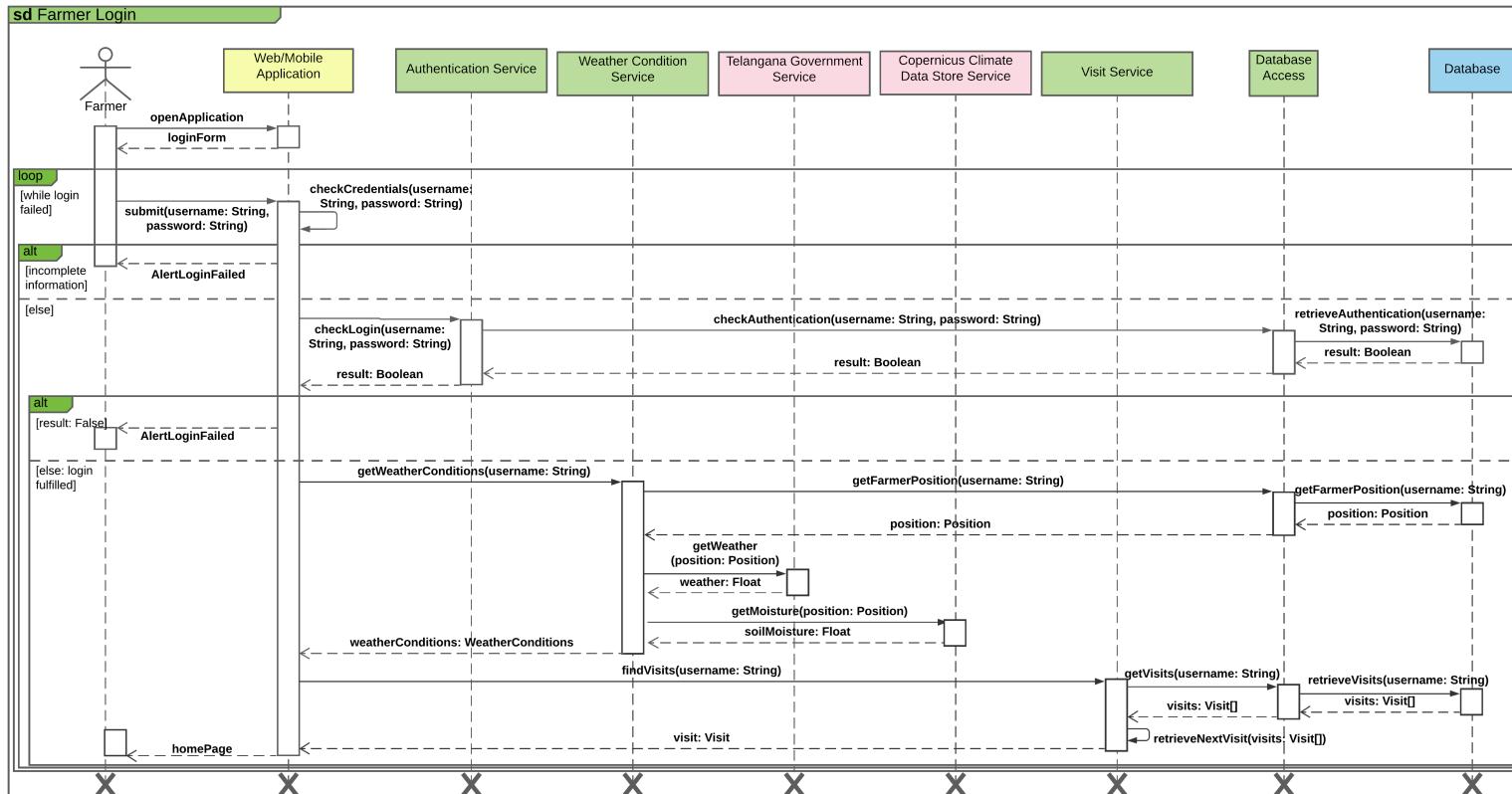


Figure 2.7: Farmer logs in

2.4.3 Farmer inserts data about his/her production

This sequence diagram shows the procedure of inserting data about production executed by a farmer who is already registered and logged in.

After having opened DREAM application on his device, the farmer clicks on *Insert* icon in the homepage. Web/MobileApplication returns a form to be filled in. Once the farmer submits it Web/MobileApplication checks whether fields are missing or invalid. If it is the case, it shows an alert to the farmer. If it is not the case, the PerformanceService is called to compute the performance of the farmer.

Farmer's performance score is affected by weather conditions, for this reason it is necessary to retrieve farmer's position and his last performance date to define the time interval for which weather and soil moisture data must be retrieved. PerformanceService calls the DatabaseAccess. Database access provides access to the Database, where all the farmers' performances and positions are stored persistently. Farmer's last performance date and position are retrieved and returned eventually to PerformanceService.

The latter component is also able to get quantity of water consumed by the farmer calling TelanganaWaterIrrigationService. The result is returned back and a function is called by PerformanceService to add it to the data inserted by the farmer. At this moment, two functions are called and propagated to two different external APIs (TelanganaGovernmentService and CopernicusClimateDataStoreService) to retrieve weather and soil moisture.

PerformanceService has now all necessary data to compute performance score, for which a proper function is called.

Eventually, farmer's performance is stored in the Database through a request which is propagated through DatabaseAccess to Database. A result is return following the same path. If it is false the process must be repeated.

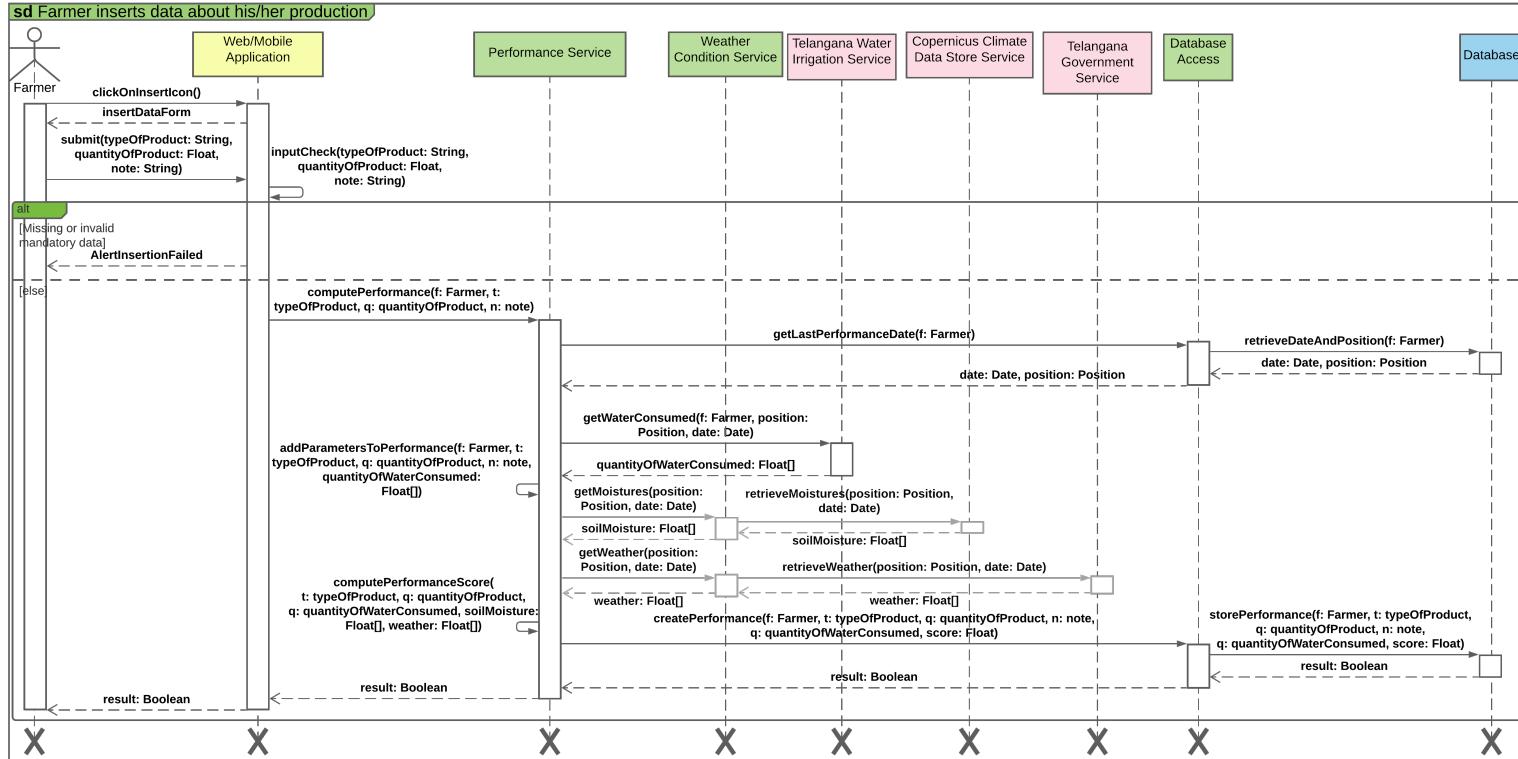


Figure 2.8: Farmer inserts data about his/her production

2.4.4 DREAM sends suggestions to farmer

This sequence diagram shows the procedure of DREAM sending periodic suggestions to a farmer who is already registered. Suggestions are computed by the system and affected by weather conditions regarding farmer's position and his/her last performance.

A request to compute suggestion is called by Web/MobileApplication and propagated to SuggestionService component which calls a function to get data needed to make the computation. For this reason the request is propagated through DatabaseAccess to Database. Farmer's last performance and position are eventually returned to SuggestionService. The latter component propagates two requests to get weather and soil moisture to WeatherConditionService, which is in charge to retrieve those data from external APIs (TelanganaGovernmentService and CopernicusClimateDataStoreService). Data are returned to SuggestionService which is now able to compute the suggestion calling a proper function.

Eventually, the suggestion is stored in the Database and is returned following the same path to Web/Mobile Application, which is able to send a notification to the farmer.

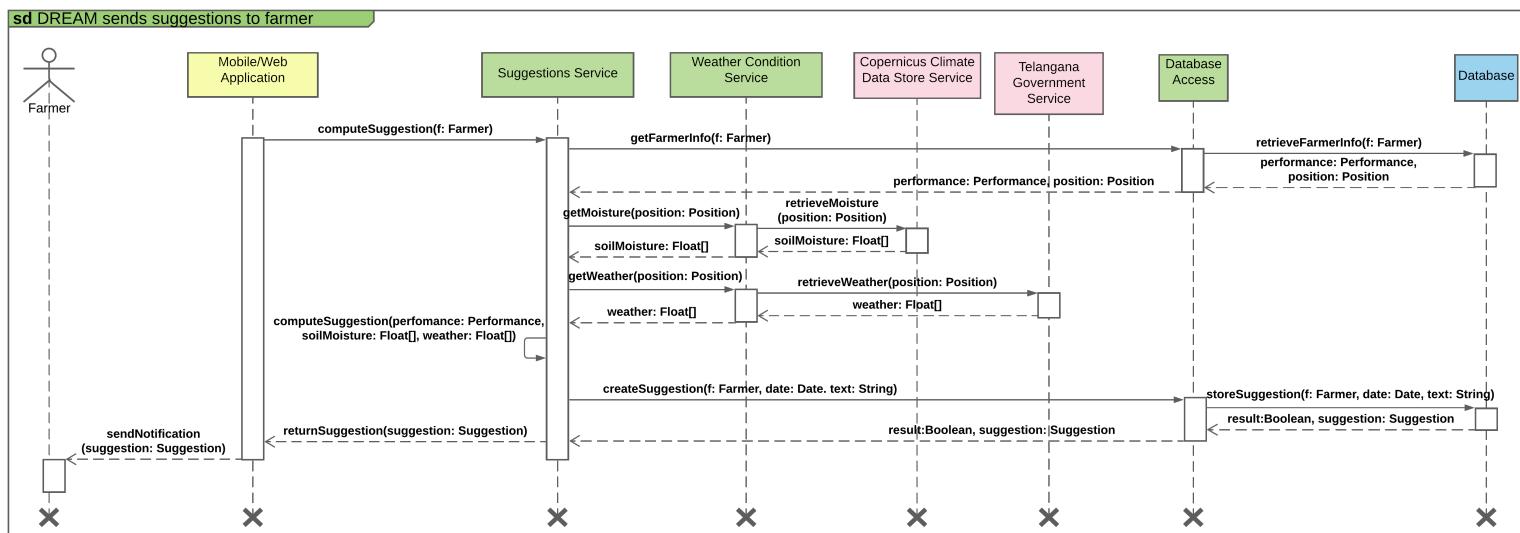


Figure 2.9: DREAM sends suggestions to farmer

2.4.5 Farmer searches for a thread in the discussion forum

This sequence diagram shows the procedure of a farmer, already registered and logged in, searching for a thread on the discussion forum.

He/she clicks on *discussion forum* icon. To return to the farmer the related page, Web/MobileApplication propagates a request to get all discussion forum's threads and related posts to DiscussionForumService which is able to retrieve them from the Database through DatabaseAccess.

Once the *discussion forum* page is shown to the farmer, he/she clicks on *search* icon and a form is returned by Web/MobileApplication. The farmer inserts the topic in the appropriated field and Web/MobileApplication calls a function to get all threads related to that topic. The request is propagated to DiscussionForumService which is in charge of finding those threads and their related posts. The result and found threads with related posts are returned to Web/MobileApplication following the same path. If result is false no thread was found and a message is sent to the user. If the result is true all found threads with related posts are shown to the farmer.

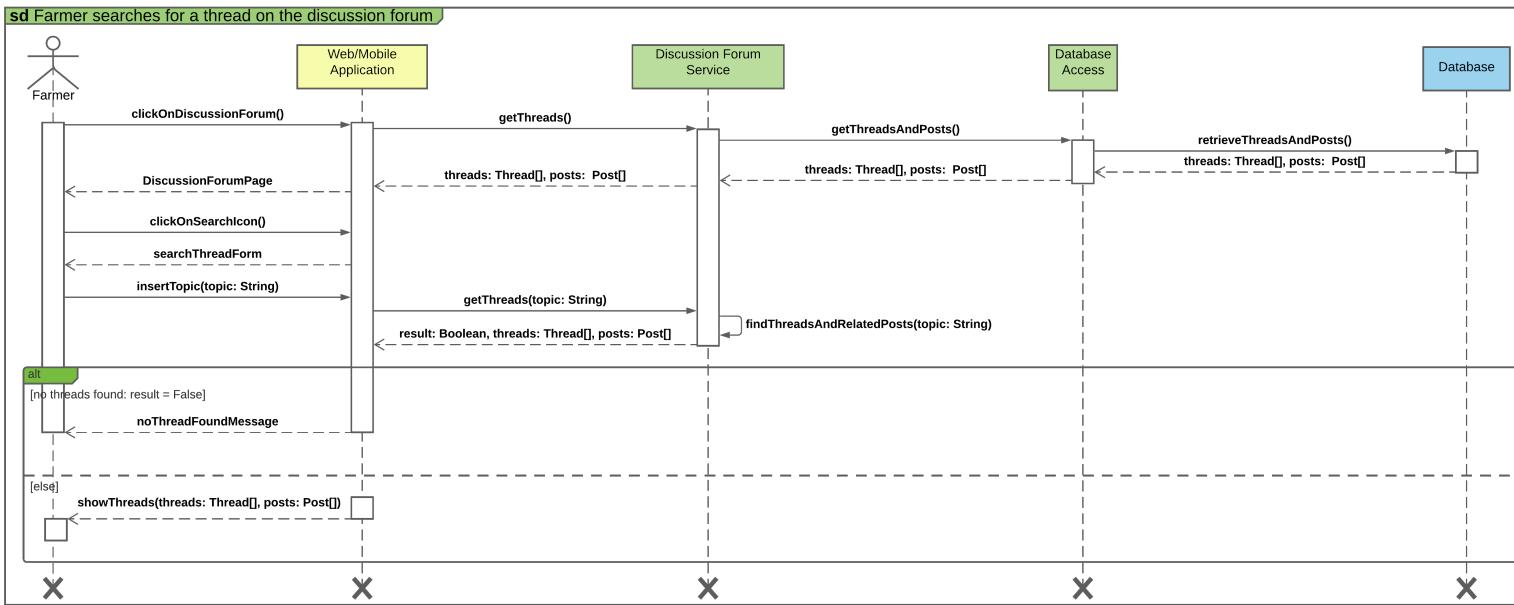


Figure 2.10: Farmer searches for a thread in the discussion forum

2.4.6 Farmer creates a new thread in the discussion forum

This sequence diagram shows the procedure of a farmer, already registered and logged in, opening a thread on the discussion forum.

He/she clicks on *discussion forum* icon. To return to the farmer the related page, Web/MobileApplication propagates a request to get all discussion forum's threads and related posts to DiscussionForumService which is able to retrieve them from the Database through DatabaseAccess. Once the *discussion forum* page is shown to the farmer, he/she clicks on *new thread* button and a form is returned by Web/MobileApplication. The farmer inserts the topic and the text in the appropriated fields and Web/MobileApplication calls a function to store the new thread. The request is propagated to DiscussionForumService which is in charge of adding additional information to the thread before propagating the request to Database through DatabaseAccess. The result is returned to Web/MobileApplication following the same path. If result is false the operation must be repeated.

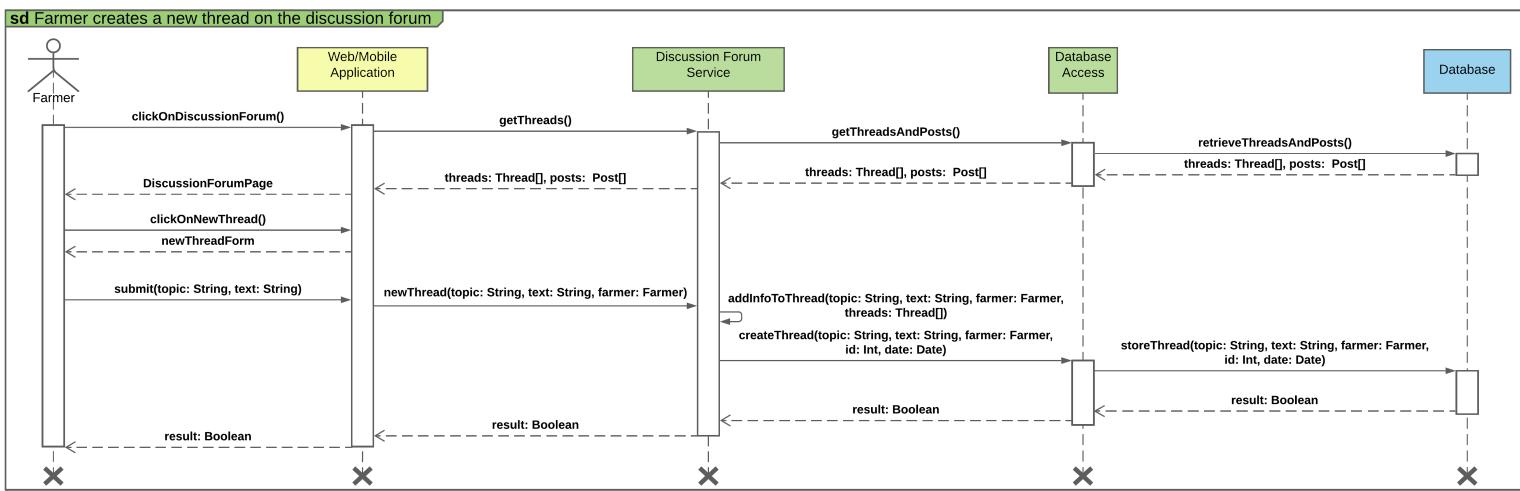


Figure 2.11: Farmer creates a new thread in the discussion forum

2.4.7 Farmer replies in the discussion forum

This sequence diagram shows the procedure of a farmer, already registered and logged in, replying on the discussion forum.

He/she clicks on *discussion forum* icon. To return to the farmer the related page, Web/MobileApplication propagates a request to get all discussion forum's threads and related posts to DiscussionForumService which is able to retrieve them from the Database through DatabaseAccess. Once the *discussion forum* page is shown to the farmer, he/she clicks on a thread whose details are returned by Web/MobileApplication after a function to find related posts is called. The farmer clicks on *comment* button and a form is returned. He/she inserts the text of the post and Web/MobileApplication calls a function to store the new post. The request is propagated to DiscussionForumService which is in charge of adding additional information to the post before propagating the request to Database through DatabaseAccess. The result is returned to Web/MobileApplication following the same path. If result is false the operation must be repeated. If the result is true the process is successfully completed and a message is sent to the farmer who created the post while a notification is sent to the one that has created the related thread.

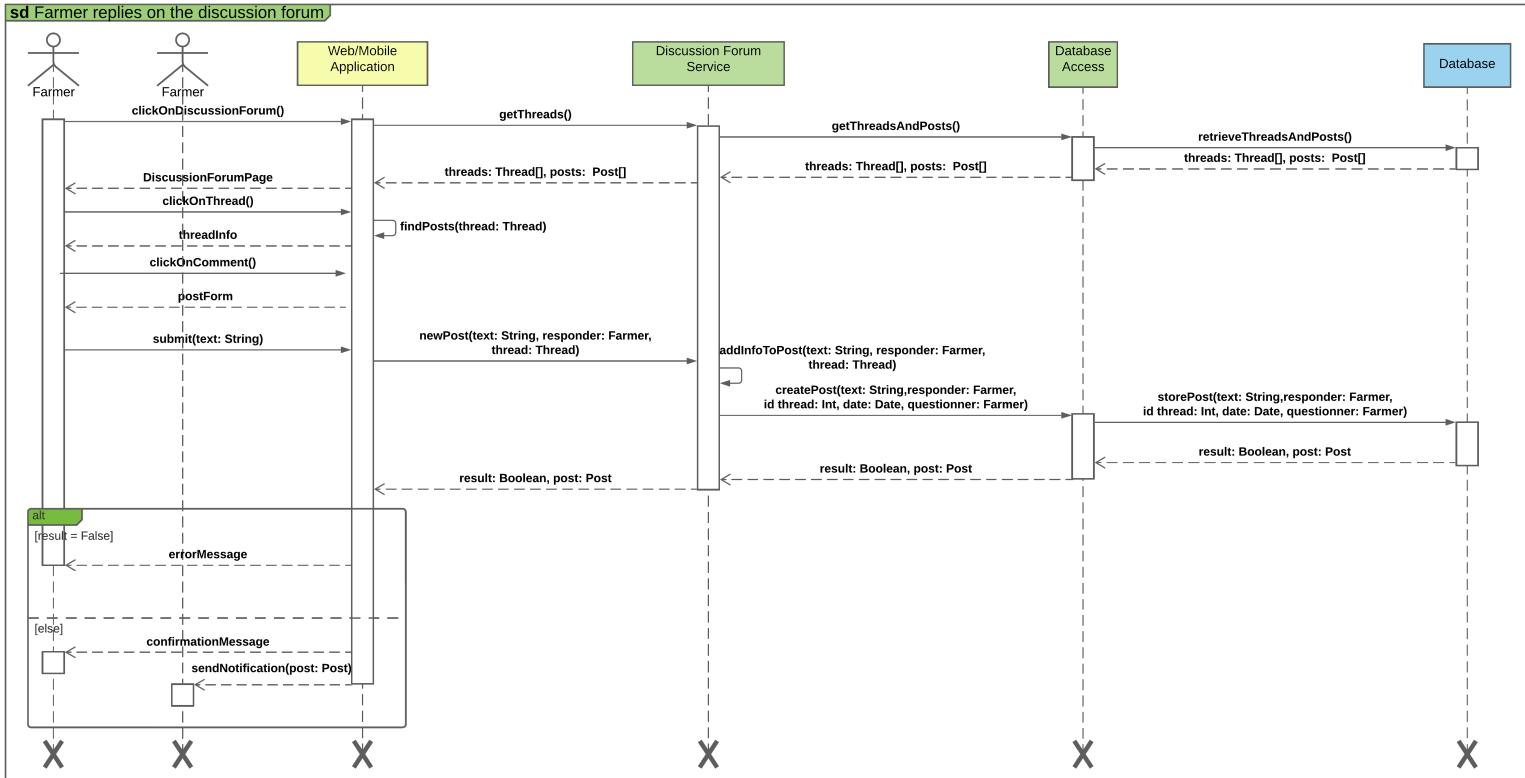


Figure 2.12: Farmer replies on the discussion forum

2.4.8 Farmer creates a help request

This sequence diagram shows the procedure of a farmer, already registered and logged in, creating a help request.

He/she clicks on *help request* icon. To return to the farmer the related page, Web/MobileApplication propagates a request to get all help requests and related help responses related to the farmer to HelpRequestService which is able to retrieve them from the Database through DatabaseAccess. Once the *help request* page is shown to the farmer, he/she clicks on *create help request* button and a form is returned by Web/MobileApplication. The farmer inserts the text in the appropriated field and checks well performing farmers as recipient if he/she wants to. Web/MobileApplication calls a function to store the new help request. The request is propagated to HelpRequestService which is in charge of adding additional information to the help request before returning it following the same path to show its details to the farmer. The help request is then stored in the Database. The result and the help request are returned to Web/MobileApplication following the same path. If result is false the operation must be repeated. If it is true a notification is always sent to the agronomist responsible of the mandal which the farmer is located in. If well performing farmers are also checked as recipient a function is called by PerformanceService to get well performing farmers given all farmers' performances. The farmers retrieved are returned to Web/MobileApplication which is now able to send them a notification about the new help request received.

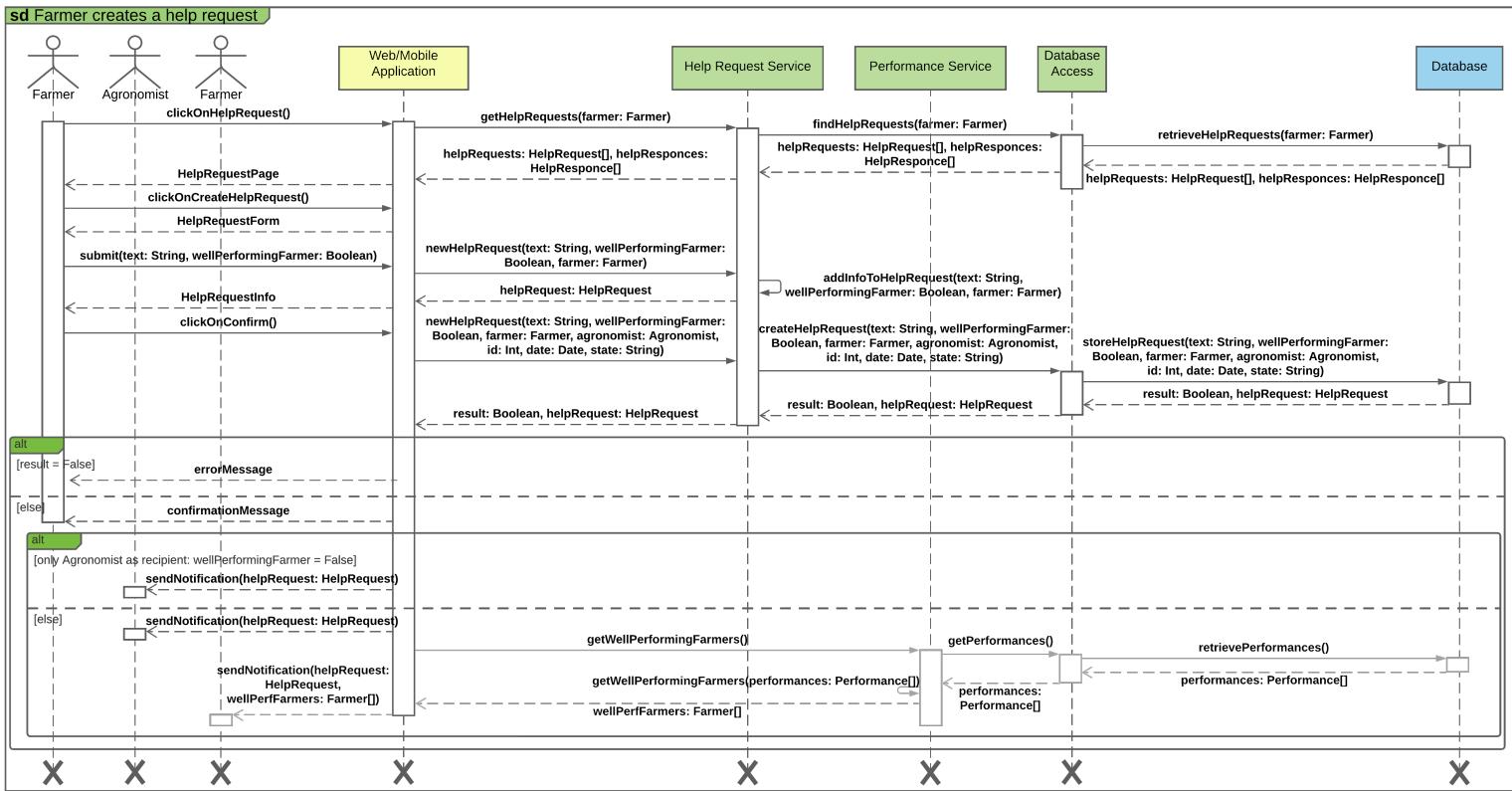


Figure 2.13: Farmer creates a help request

2.4.9 Agronomist replies to a help request

This sequence diagram shows the procedure of an agronomist, already registered and logged in, replying to a help request. The same functionality can be performed by a well performing farmer if he/she is checked as recipient in the help request.

The agronomist clicks on *help request* icon. To return to the agronomist the related page, Web/MobileApplication propagates a request to get all help requests related to the agronomist to HelpResponseService which is able to retrieve them from the Database through DatabaseAccess. Once the *help request* page is shown to the agronomist, he/she clicks on *reply* button and a form is returned by Web/MobileApplication. The agronomist inserts the text in the appropriated field and Web/MobileApplication calls a function to store the new help response. The request is propagated to HelpResponseService which is in charge of adding additional information to the help response. The help response is then stored in the Database. The result and the help response are returned to Web/MobileApplication following the same path. If result is false the operation must be repeated. If it is true a notification is sent to the farmer who made the help request and a confirmation message is sent to the agronomist.

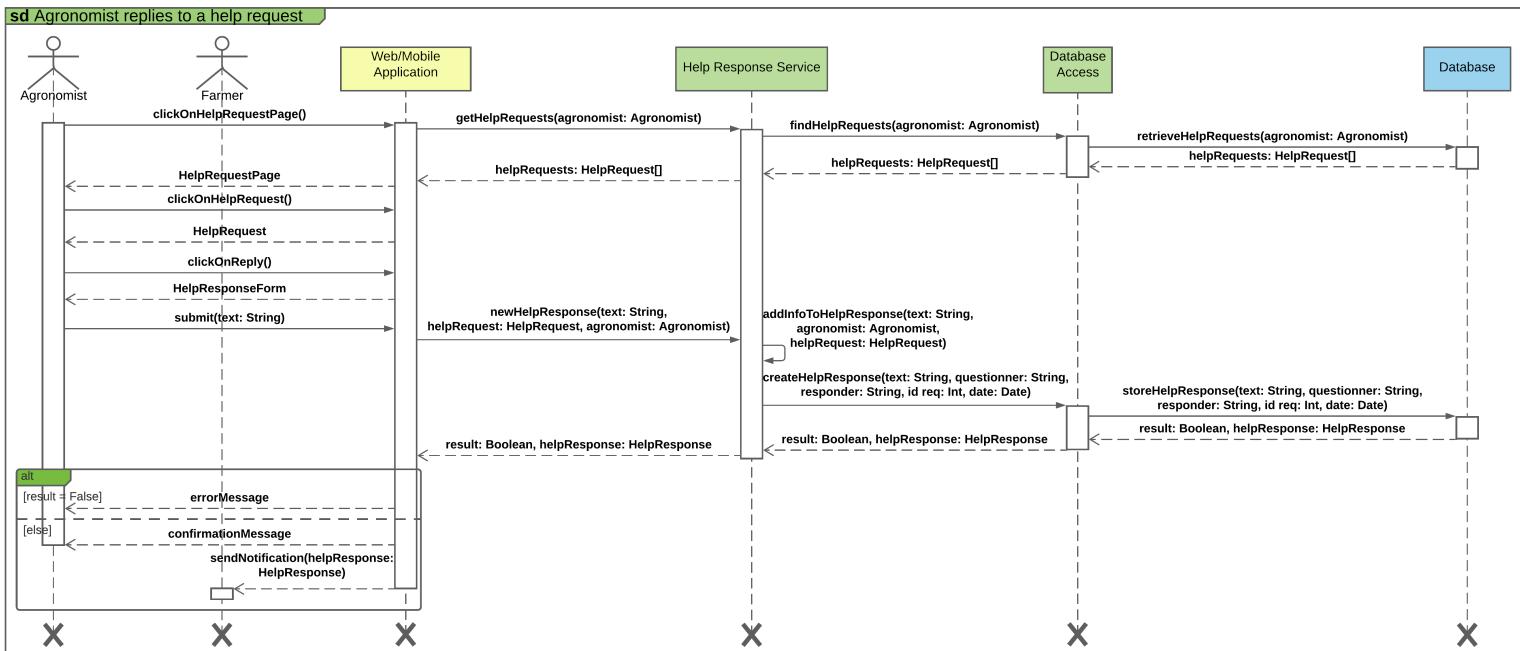


Figure 2.14: Agronomist replies to a help request

2.4.10 Farmer solves a help request

This sequence diagram shows the procedure of a farmer, already registered and logged in, solving a help request.

The farmer clicks on *help request* icon. To return to the farmer the related page, Web/MobileApplication propagates a request to get all help requests and help responses related to the farmer to HelpRequestService which is able to retrieve them from the Database through DatabaseAccess. Once the *help request* page is shown to the farmer, he/she clicks on *my help requests* button to visualize those created by him/her and the related page is returned by Web/MobileApplication. The farmer solves a help request clicking on the related button and Web/MobileApplication calls a function to update the status of the help request. The request is propagated to HelpRequestService and then the help request state is updated in the Database. The result is returned to Web/MobileApplication following the same path. If result is false the operation must be repeated.

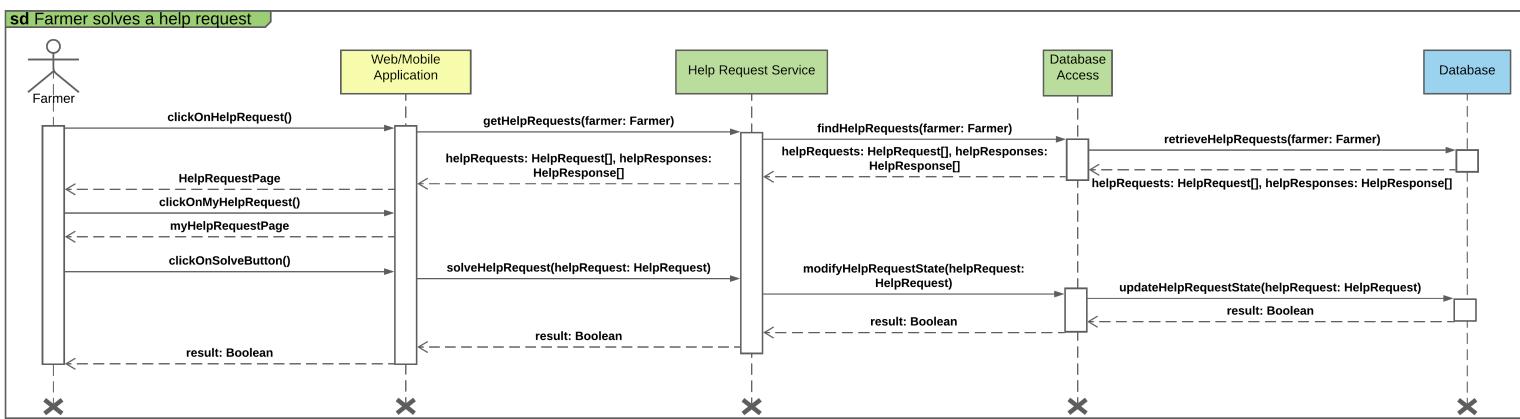


Figure 2.15: Farmer solves a help request

2.4.11 Agronomist visualizes weather conditions

This sequence diagram shows the procedure of an agronomist, already registered and logged in, visualizing weather conditions.

The agronomist clicks on *weather* icon. He/she can optionally select the date (from the current day up to seven days) for which he/she wants to visualize data. By default, if he/she doesn't select anything, the system shows him/her weather conditions for the current day. To return to the agronomist the related page, Web/MobileApplication propagates a request to get the weather map of his/her mandal to WeatherForecastsService which is able to retrieve weather and soil moisture data from external APIs (TelanganaGovernmentService and CopernicusClimateDataStoreService). Then, data retrieved are returned to WeatherForecastsService which is in charge also of processing the mandal weather map interacting with GoogleMapsAPI. Eventually, the map is shown to the agronomist.

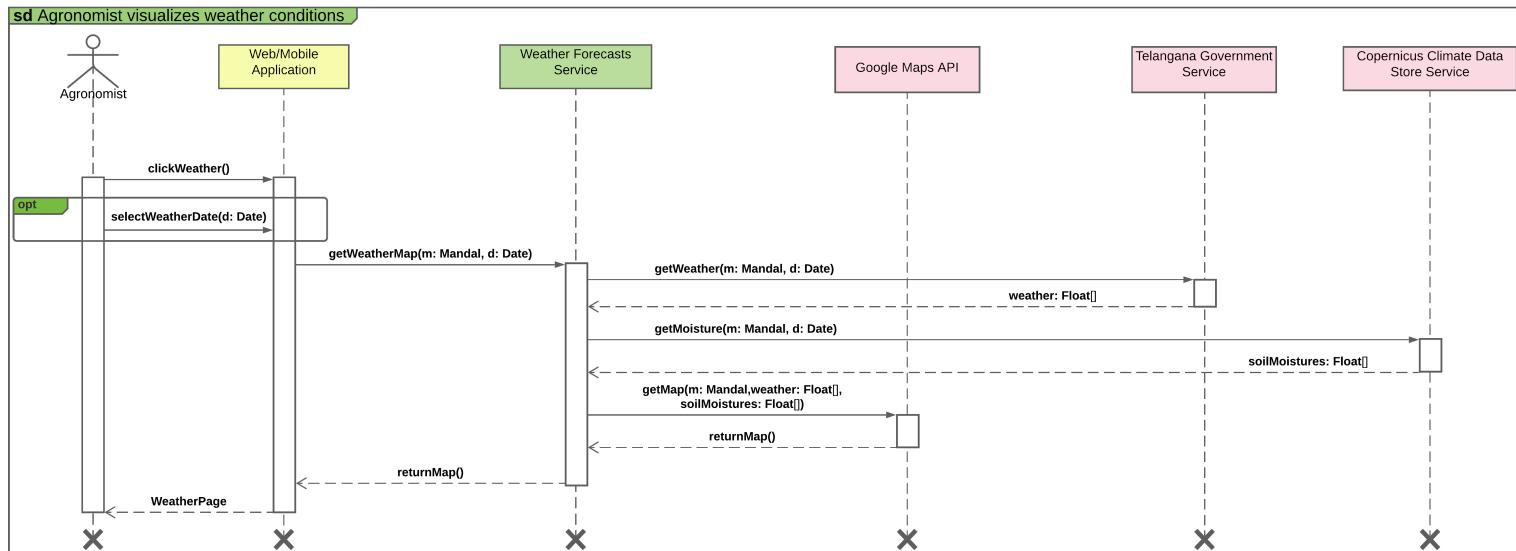


Figure 2.16: Agronomist visualizes weather conditions

2.4.12 Agronomist visualizes and analyzes farmers' data

This sequence diagram shows the procedure of an agronomist, already registered and logged in, visualizing and analyzing his/her mandal's farmers' data in the homepage of the application.

The agronomist opens the application. Web/MobileApplication propagates two requests, one to TableService to get a table of farmers' performances in descending order and the other one to MandalMapService to get the mandal map on which are displayed farmers's performances scores. For both services it is necessary to retrieve from the Database mandal's farmers' data (including performances and positions). Those data are returned to TableService and to MandalMapService. The latter component is in charge of processing the map to be displayed to the agronomist, for this reason it interacts with GoogleMapAPI external service. Eventually the *homepage* is shown to the agronomist.

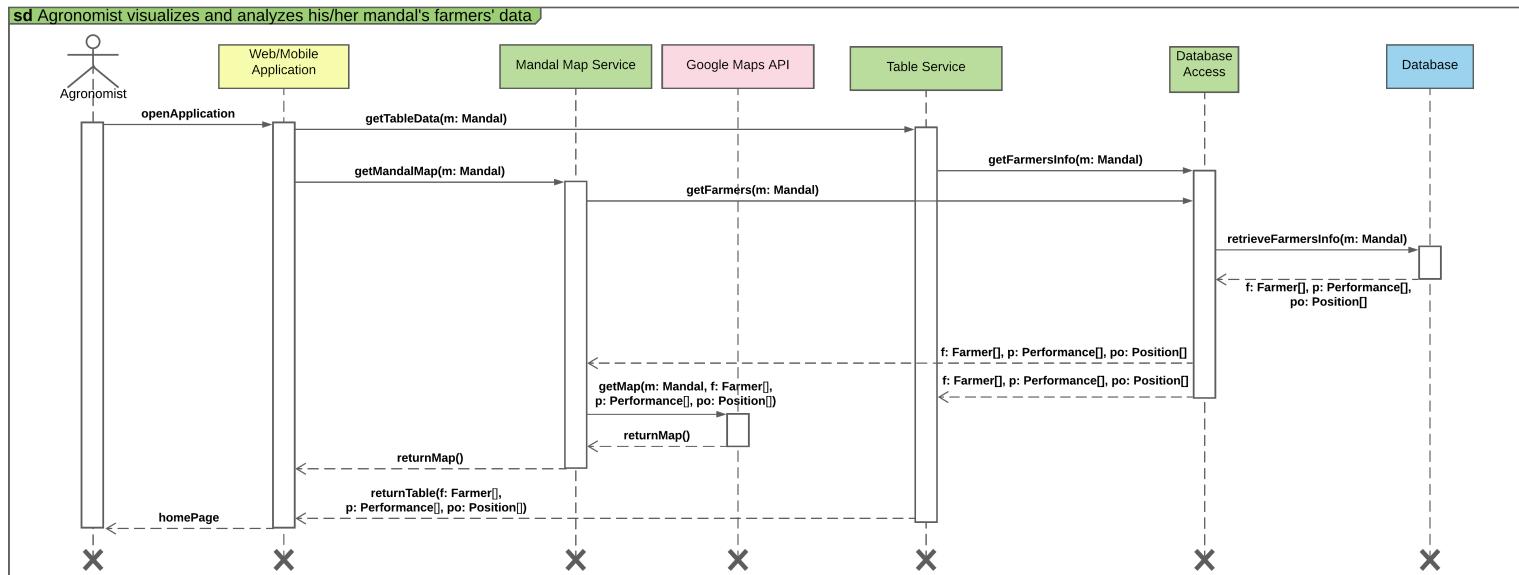


Figure 2.17: Agronomist visualizing and analyzing farmers' data

2.4.13 Agronomist updates his/her daily plan

This sequence diagram shows the procedure of an agronomist, already registered and logged in, updating his/her daily plan.

The agronomist clicks on *daily plan* icon. To return to the agronomist the related page, Web/MobileApplication propagates a request to get all daily plans and visits related to the agronomist to DailyPlanService which is able to retrieve them from the Database through DatabaseAccess. Once the *daily plan* page is shown to the agronomist, he/she clicks on *update* button. The update form is returned by Web/MobileApplication. The agronomist fills out the form and a function is called by Web/MobileApplication to get the visit corresponding to date and hour inserted by the agronomist. The function is propagated to DailyPlanService which is in charge of finding the visit and returns it following the same path. If result is false no visit was found and a new visit is created and stored in the Database. If result is true a visit with the date and hour inserted by the agronomist already exists, for this reason it is updated in the database according to the information entered by the user. In both cases, the visit's status is set into scheduled and the corresponding daily plan's state is change into updated. Eventually the result and the visit are returned following the same path and a notification is sent to the farmer whose visit has been created/changed.

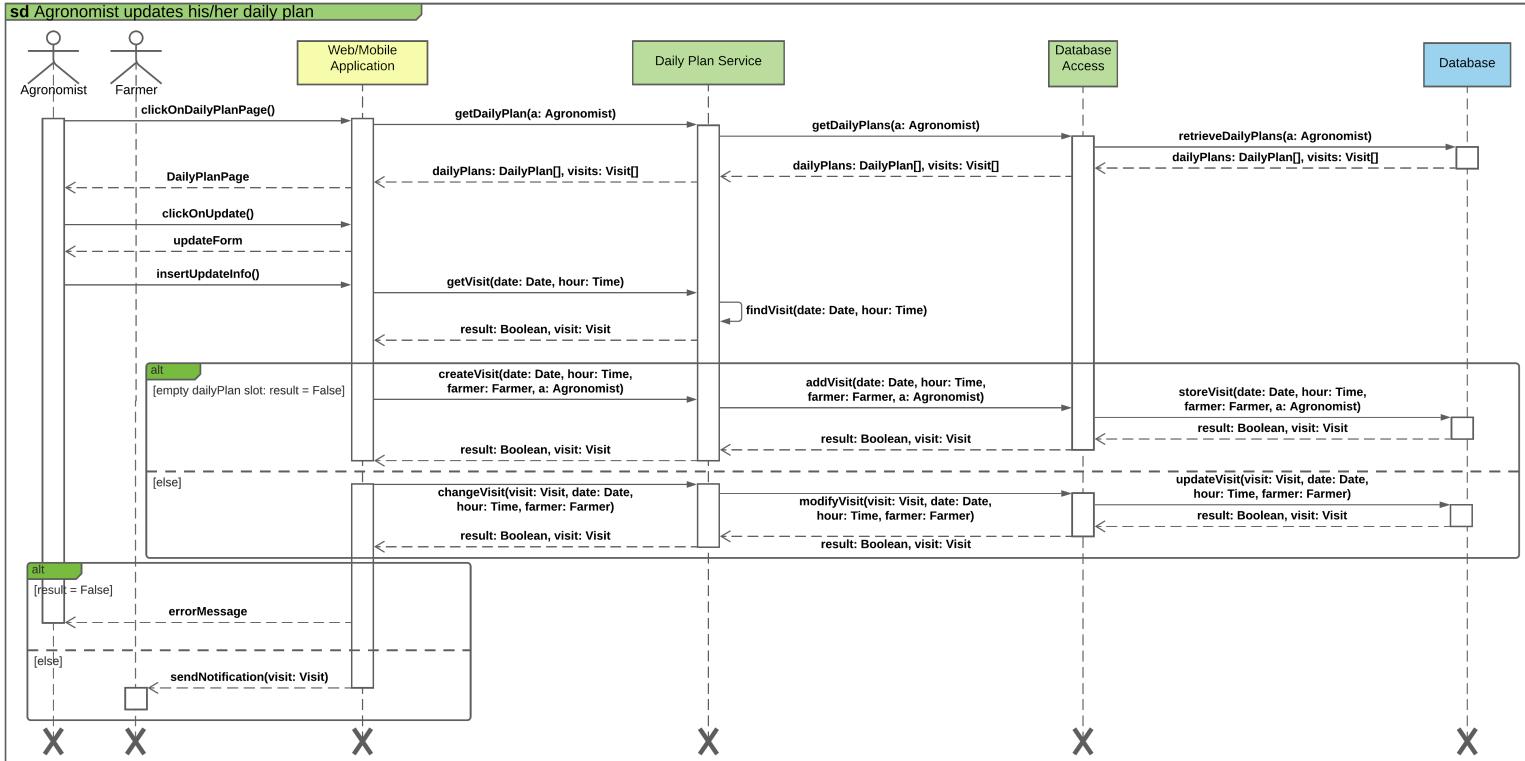


Figure 2.18: Agronomist updates his/her daily plan

2.4.14 Agronomist confirms his/her daily plan

This sequence diagram shows the procedure of an agronomist, already registered and logged in, confirming his/her daily plan.

The agronomist clicks on *daily plan* icon. To return to the agronomist the related page, Web/MobileApplication propagates a request to get all daily plans and visits related to the agronomist to DailyPlanService which is able to retrieve them from the Database through DatabaseAccess. Once the *daily plan* page is shown to the agronomist, he/she clicks on a certain daily plan page and a function to retrieve the selected daily plan and related visits is called by Web/MobileApplication and propagated to DailyPlanService which is in charge of finding the selected daily plan. The result is returned following the same path and daily plan details are shown to the agronomist. He/she clicks on *confirm* button and the confirmation form is returned by Web/MobileApplication. The agronomist optionally inserts deviations from the daily plan in the appropriate field and then submits. Then, a function is called and propagated by Web/MobileApplication to change the state of the daily plan into confirmed and related visits' status into done in the Database. Eventually the result is returned following the same path. If result is false the process must be repeated.

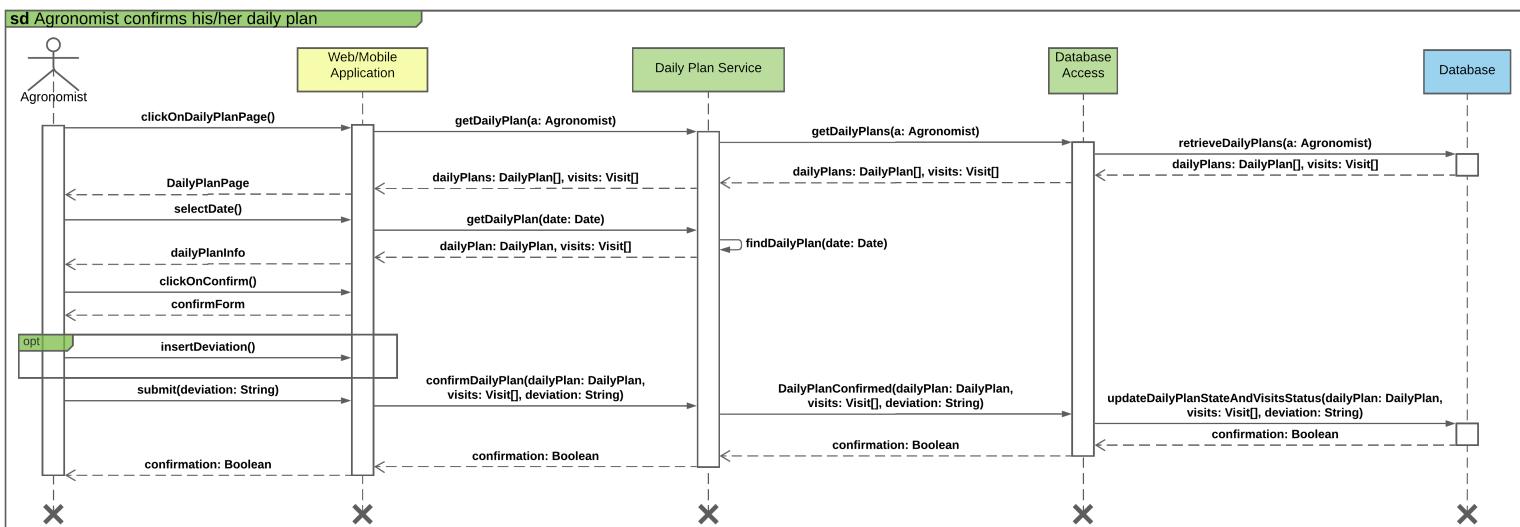


Figure 2.19: Agronomist confirms his/her daily plan

2.4.15 Policy Maker visualizes and analyzes farmers' data

This sequence diagram shows the procedure of a policy maker, already registered and logged in, visualizing and analyzing farmers' data in the homepage of the application.

The policy maker opens the application and he/she can select map and time chart options. If no option is modified the system process time chart and map automatically with default parameters. Web/MobileApplication propagates two requests, one to TimeChartService to get a chart of farmers' performances trend and the other one to MapService to get the Telangana map on which are displayed farmers's performances. For both services is necessary to retrieve from the Database farmers' data(performances and positions). Those data are returned to TimeChartService and to MapService. Both of them compute data according to the parameters selected by the policy maker or those acquired by default, then they return the result. In particular MapService is in charge of processing the map on which will be displayed farmers' performances, to do so it interacts with GoogleMapsAPI external service. Eventually the *homepage* is shown to the policy maker.

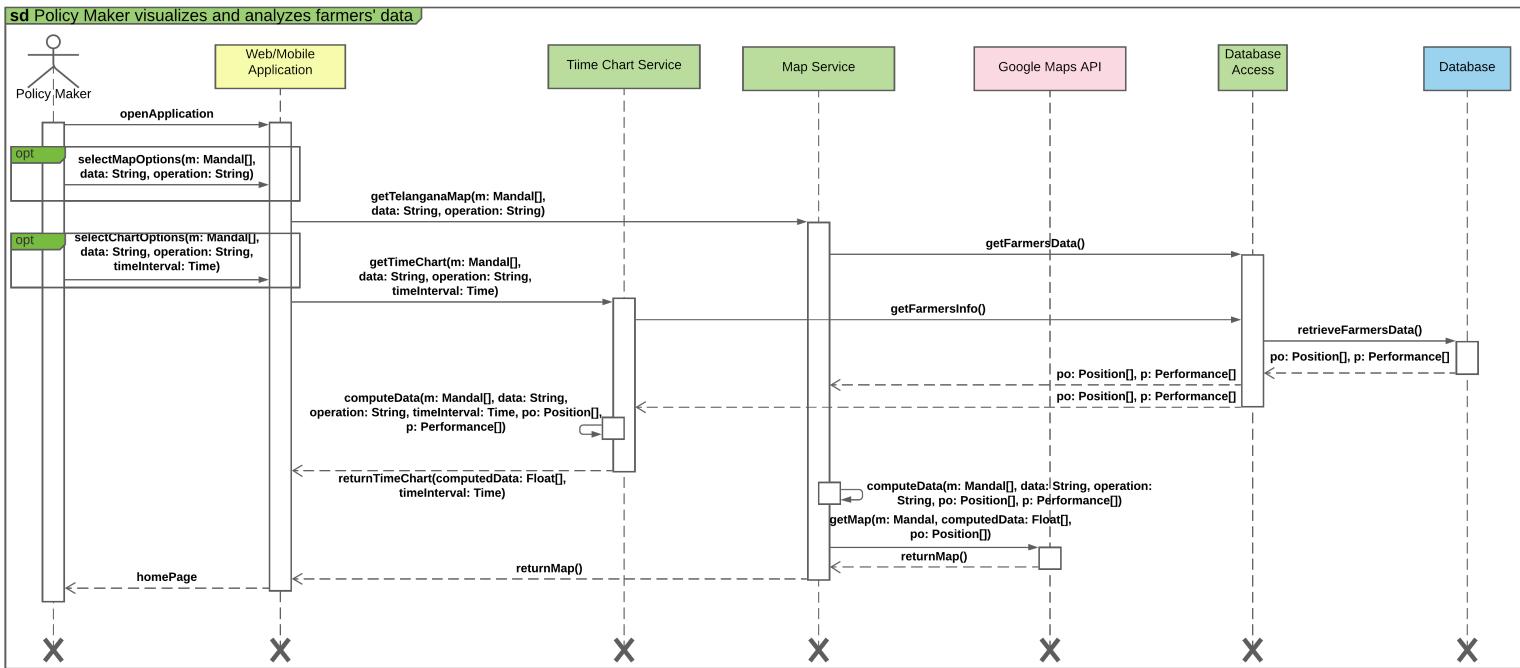


Figure 2.20: Policy Maker visualizes and analyzes farmers' data

2.5 Component Interfaces

As shown in *Section 2.4*, many method are invoked on the interfaces. Their interactions guarantee the execution of the most important processes required to fulfill DREAM's goals.

The following diagram is presented to better understand how interfaces interact with each other.

The methods listed in the diagram are a logical representation of what component interfaces have to guarantee the users.

Certainly, considering a future implementation, they will be subjected to many modifications, according to choices and needs of developers.

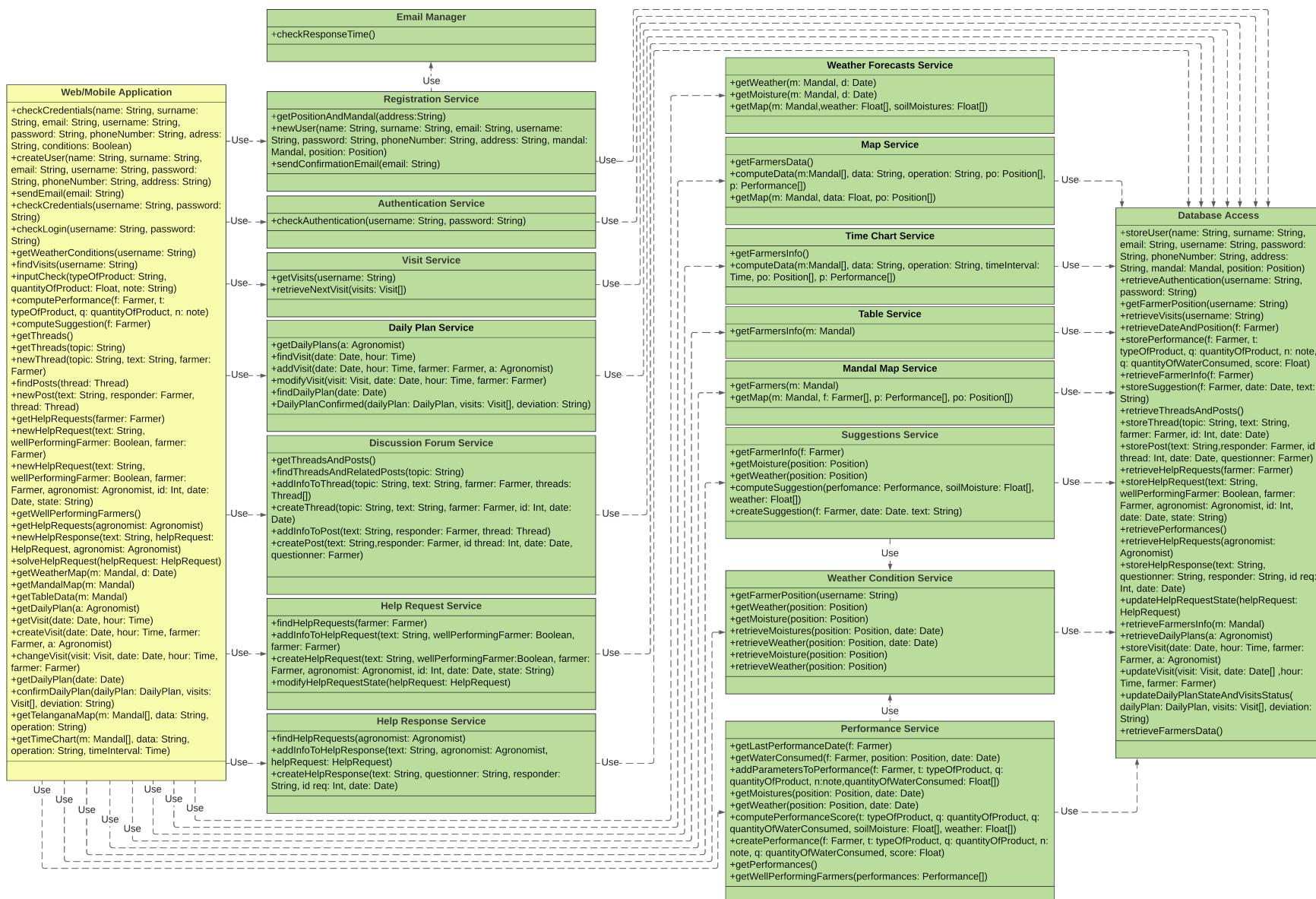


Figure 2.21: Component Interfaces diagram

2.6 Selected Architectural Style and Patterns

This section clarifies the architectural style and architectural pattern adopted in the design of the DREAM system. The main difference between the two is that an architectural style is the application design at the highest level of abstraction, it is a name given to a recurrent architectural design; while an architectural pattern is a way to implement an architectural style, it is a way to solve a recurring architectural problem related to it.

2.6.1 Architectural Style: Client-Server

In the design of the DREAM application a three tier architecture has been used. It is a client-server architecture in which the functional process logic, data access, computer data storage and user interface are developed and maintained as independent modules on three levels:

- a presentation tier;
- a logic tier;
- a data tier.

In this type of architecture there is the thin client, that is the device that requests the resource, equipped with a user interface in charge of the presentation-level functions. Then, there is the application server, also called middleware, in charge of providing the resource and communicating with the server database, which stores the data used by the application.

The main advantage of the three-tier architecture is the logical and physical separation of functionality. This allows each tier to run on a separate operating system and server platform that best suit its functional requirements.

Compared to one or two tier architecture, this allows for **faster development** as each tier can be developed simultaneously by several teams. Furthermore, this separation of levels allows programmers to use the latest and best languages and tools for each level.

Moreover, any tier can be scaled independently of the others as needed and if one tier is interrupted, it will be less likely to impact the availability or performance of the other tiers, so this architecture also has a positive impact on **scalability** and **reliability** as well.

Since the presentation layer and the data layer cannot communicate directly, a well-designed application layer can function as a kind of internal firewall, preventing some malicious attacks and ensuring **greater security**.

2.6.2 Architectural Pattern: Model-View-Controller Structure

DREAM application can be structured with the Model-View-Controller (MVC) pattern. The MVC structure consists of three building blocks: model, view, and controller.

- the **Model** provides the methods to access the data useful for the application;
- the **View** displays the real user interface for the presentation of the data contained in the model and deals with the interaction with users;
- the **Controller** receives the user's commands through the view and implements them by accessing the Model and defining the corresponding View to be presented

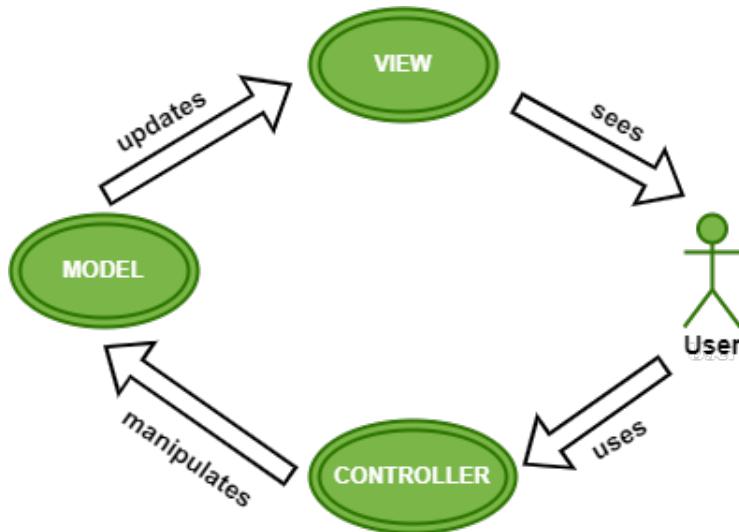


Figure 2.22: Model-View-Controller structure

This pattern was chosen primarily because application development becomes fast. In fact, it is easier for multiple developers to collaborate and work together given the nature of the architecture itself. As a result, it also becomes easier to update the application and debug as we have multiple levels properly written in the application.

2.7 Other Design Decisions

This section describes design decisions that have not been described in the other sections of the document in order to clarify their operation and purpose.

2.7.1 ID code for policy maker and agronomist from external database

DREAM application implies the interaction with two particular kinds of user: the policy maker, an institutional figure, and the agronomist, a professional figure. Agronomists and policy makers must be authorized through a personal and unique ID code to be able to exploit DREAM's functionalities according to their role.

It is assumed that those unique id codes are created by Telangana Government and stored persistently in one of its databases accessible from the application only thanks to the external API TelanganaGovernmentService.

This means that DREAM has only the permission to visualize the data in the external database to check that the entered id codes are correct, but it has no permission to modify or update or delete it.

2.7.2 Internal database replication

As showed in *Figure 2.1*, the replication of the database that contains the persistent data of the system is used with the aim of improving the reliability and performance of the DREAM application.

This technique consists of copying a database instance to exactly another location. It is used in distributed database management systems environments where a single database must be used and updated in multiple locations at the same time.

A full replica of the entire database is created at each site of the distributed system. This choice improves system availability because it can continue to work properly as long as at least one site is active.

Incremental backup method is expected, with only the changes made after the last backup archived instead of backing up the whole target.

Since a single update must be performed on different databases to keep the copies consistent, the update process of the databases can be slow. However, it is necessary because the DREAM application is based precisely on the collection of data that are fundamental for the analysis of farmers' performances.

Chapter 3

User Interface Design

Some interfaces, for both WebApp and MobileApp, have already been presented in RASD document.

Following figures display a graph, called *flow graph*, that represents the main screens of DREAM from MobileApp perspective for farmer and agronomist actors.

Farmer

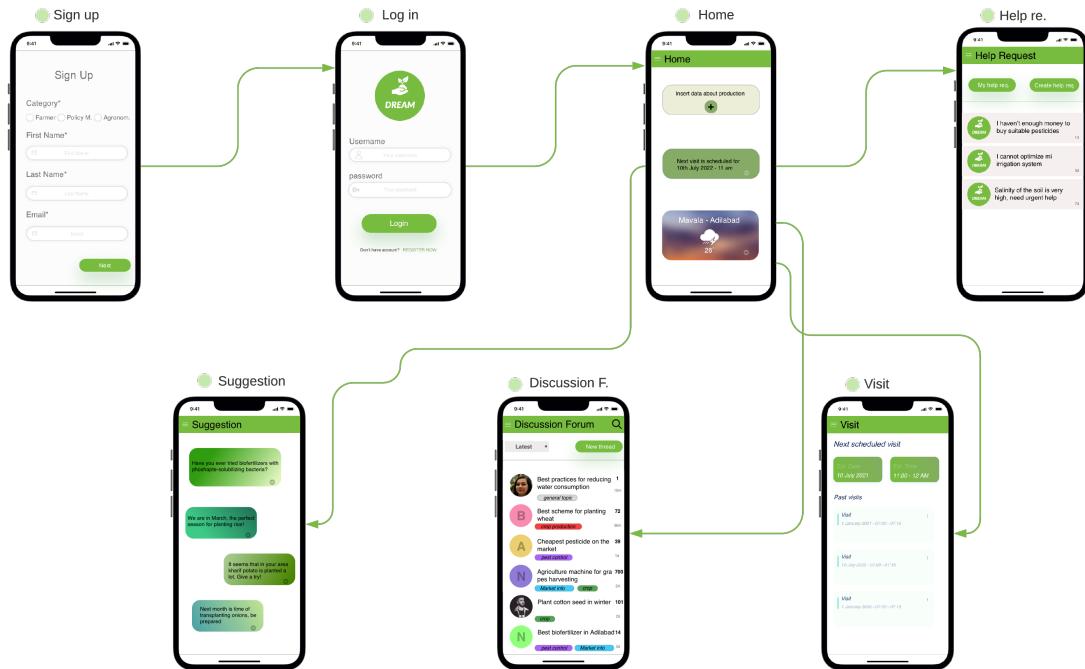


Figure 3.1: Farmer Interfaces

Agronomist

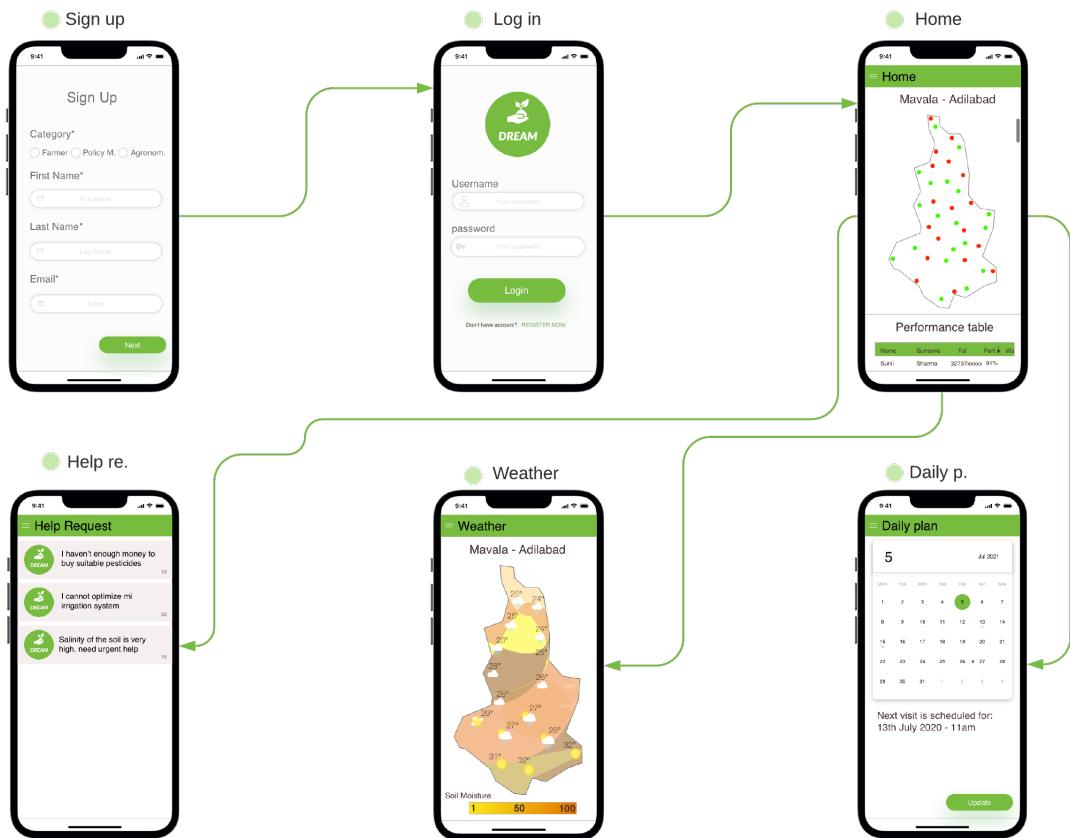


Figure 3.2: Agronomist Interfaces

Chapter 4

Requirement Traceability

In order to check whether all requirements listed in the RASD have been considered designing DREAM application, they are mapped below with the related components described in *Section 2.2*.

Furthermore, a good mapping allows to verify if a correct modularity is reached, which allows to manage a complex software for implementation and maintenance purposes.

R.1 Unregistered user must be able to register to DREAM

- MobileApplication/WebApplication
- RegistrationService
- EmailManager
- GoogleMapsAPI
- TelanganaGovernmentService
- DatabaseAccess
- Database

R.2 User must be able to login to DREAM by filling out the login form

- MobileApplication/WebApplication
- AuthenticationService
- DatabaseAccess
- Database

R.3 The system must be able to associate each farmer to the corresponding mandal according to his/her position

- MobileApplication/WebApplication
- RegistrationService

- GoogleMapsAPI
- DatabaseAccess
- Database

R.4 Farmer must be able to fill out the form for data insertion

- MobileApplication/WebApplication

R.5 The system must be able to add parameters to data inserted by farmers regarding their production

- MobileApplication/WebApplication
- PerformanceService
- WeatherConditionService
- TelanganaWaterIrrigationService
- CopernicusClimateDataStoreService
- TelanganaGovernmentService
- DatabaseAccess
- Database

R.6 The system must be able to return whether a data insertion has been done correctly by the user or not

- MobileApplication/WebApplication

R.7 The system must be able to classify farmers into bad and well performing farmers according to their performance score value

- PerformanceService
- DatabaseAccess
- Database

R.8 The system must be able to show the map with performance score regarding farmers

- MobileApplication/WebApplication
- MapService
- MandalMapService
- GoogleMapsAPI
- DatabaseAccess
- Database

R.9 The policy maker must be able to select options which he/she wants to update the map with

- MobileApplication/WebApplication

R.10 The system must be able to show the updated map according to the option chosen by policy maker.

- MobileApplication/WebApplication
- MapService
- GoogleMapsAPI
- DatabaseAccess
- Database

R.11 The policy maker must be able to select options for the time chart

- MobileApplication/WebApplication

R.12 The system must be able to show the time chart according to the option chosen by policy maker.

- MobileApplication/WebApplication
- TimeChartService
- DatabaseAccess
- Database

R.13 The system must be able to compute data according to the chosen operation

- TimeChartService
- MapService

R.14 The system must be able to show a table with farmers' data in descending order based on their performance score

- MobileApplication/WebApplication
- TableService
- DatabaseAccess
- Database

R.15 The system must be able to show the map of mandal's agronomist with weather conditions and soil moisture of the current date

- MobileApplication/WebApplication

- WeatherForecastsService
- GoogleMapsAPI
- TelanganaGovernmentService
- CopernicusClimateDataStoreService

R.16 The agronomist must be able to visualize weather forecasts up to seven days after the current date

- MobileApplication/WebApplication
- WeatherForecastsService
- GoogleMapsAPI
- TelanganaGovernmentService
- CopernicusClimateDataStoreService

R.17 The system must be able to show the daily plan

- MobileApplication/WebApplication
- DailyPlanService
- DatabaseAccess
- Database

R.18 The agronomist must be able to update the daily plan

- MobileApplication/WebApplication
- DailyPlanService
- DatabaseAccess
- Database

R.19 The agronomist must be able to confirm the daily plan

- MobileApplication/WebApplication
- DailyPlanService
- DatabaseAccess
- Database

R.20 The system must be able to show weather conditions and soil moisture regarding farmer's position

- MobileApplication/WebApplication
- WeatherConditionService
- TelanganaGovernmentService
- CopernicusClimateDataStoreService

- DatabaseAccess
- Database

R.21 The farmer must be able to visualize all visits scheduled for him/her

- MobileApplication/WebApplication
- VisitService
- DatabaseAccess
- Database

R.22 The system must be able to send notifications to the farmer/agronomist

- MobileApplication/WebApplication

R.23 The farmer and the agronomist must be able to visualize notifications' details

- MobileApplication/WebApplication

R.24 The system must be able to compute suggestions according to the information inserted by the farmer and his/her position

- MobileApplication/WebApplication
- SuggestionsService
- WeatherConditionService
- TelanganaGovernmentService
- CopernicusClimateDataStoreService
- DatabaseAccess
- Database

R.25 The farmer must be able to request for help

- MobileApplication/WebApplication
- HelpRequestService
- PerformanceService
- DatabaseAccess
- Database

R.26 The system must be able to show to well performing farmers a list of the help requests created and not yet solved with them as recipient

- MobileApplication/WebApplication

- HelpRequestService
- DatabaseAccess
- Database

R.27 The well performing farmer must be able to select a help request to read its details

- MobileApplication/WebApplication

R.28 The system must be able to show the details of the help request selected by the well performing farmers

- MobileApplication/WebApplication

R.29 The well performing farmer must be able to answer to a help request

- MobileApplication/WebApplication
- HelpRequestService
- DatabaseAccess
- Database

R.30 The system must be able to notify the agronomist when a new help request is created and not yet solved

- MobileApplication/WebApplication

R.31 The agronomist must be able to visualize notification's details

- MobileApplication/WebApplication

R.32 The system must be able to show to the agronomist a list of the help requests created and not yet solved with him/her as recipient.

- MobileApplication/WebApplication
- HelpResponseService
- DatabaseAccess
- Database

R.33 The agronomist must be able to visualize help requests' details

- MobileApplication/WebApplication

R.34 The agronomist must be able to answer to a help request

- MobileApplication/WebApplication
- HelpResponseService
- DatabaseAccess
- Database

R.35 The farmer must be able to solve a help request

- MobileApplication/WebApplication
- HelpResponseService
- DatabaseAccess
- Database

R.36 The system must be able to show a list of existing threads only visualizing the associated topic

- MobileApplication/WebApplication
- DiscussionForumService
- DatabaseAccess
- Database

R.37 The farmer must be able to visualize a certain thread with related posts

- MobileApplication/WebApplication

R.38 The farmer must be able to search for a topic on the discussion forum

- MobileApplication/WebApplication

R.39 The farmer must be able to open a thread on the discussion forum

- MobileApplication/WebApplication
- DiscussionForumService
- DatabaseAccess
- Database

R.40 The farmer must be able to answer on the discussion forum

- MobileApplication/WebApplication
- DiscussionForumService
- DatabaseAccess
- Database

Chapter 5

Implementation, Integration and Test Plan

In this chapter it is illustrated the approach that will be adopted: in *Section 5.1* and *Section 5.2* the Implementation Plan and Integration Strategy are described, while in *Section 5.3* it is presented the Testing Plan.

5.1 Implementation Plan

In order to choose which component will be implemented first, there are two main strategies that could be followed:

- Bottom-Up
- Top-Down

The approach that has been chosen for this task, and also for the Integration and Testing, is the first one, Bottom-up. First of all it permits to extend, in a future, the functionalities of DREAM without so much effort compared to Top-Down, and programmers could also perform testing and integration in parallel with the implementation in a quite easy way: when a component is implemented, programmers perform Unit testing immediately, and as the system grows through the Integration testing it is checked that the different modules interact correctly. When the entire system is completed, using this incremental process, System testing is performed, that will ensure that every requirement has been satisfied.

The main disadvantage of this strategy is that it is not possible to release a "early version" of the product, however this issue is compensated by a higher execution speed that the Bottom-Up approach ensures. In this kind of approach, in order to establish which component will be implemented first, it's important to exploit dependencies between components: if a component relies on another one, necessarily the latter must be integrated before the

former. Adopting the method described above, we start by choosing components which are completely independent from the other ones, or that, as in this case, only rely on external components.

In parallel this two components could be implemented:

- **Database** (with its subcomponent DBMS);
- **ExternalAPIs** (with its subcomponents GoogleMapsAPI, CopernicusClimateDataStoreService, TelanganaGovernmentService, TelanganaWaterIrrigationService).

Then, the next component that should be implemented is **DatabaseAccess**, as stated in previous chapters it is responsible to get data from a data source, in our case with **Database**.

Going on, **AuthenticationService** and **RegistrationService** could be implemented in parallel, since they manage login and registration to DREAM. Finally high level components, that belong to the three main actors of the application should be implemented in this order:

- **FarmerService** (with its subcomponents HelpRequestService, DiscussionForumService, PerformanceService, VisitService, WeatherConditionService, SuggestionsService);
- **AgronomistService** (with its subcomponents HelpResponseService, MandalMapService, TableService, DailyPlanService, WeatherForecastsService);
- **PolicyMakerService** (with its subcomponents TimeChartService and MapService).

In conclusion, to sum up, the components of DREAM will be implemented in the following order:

1. Database and ExternalAPIs
2. DatabaseAccess
3. AuthenticationService and RegistrationService
4. FarmerService (and its subcomponents)
5. AgronomistService (and its subcomponents)
6. PolicyMakerService (and its subcomponents)

5.2 Integration Strategy

This section describes the integration plan for the system to be developed. As stated in the previous section, each component must first undergo *Unit Testing*. The aim of this section is to verify the interactions between the modules of DREAM. Firstly components are tested individually *Unit Testing* and then combined to make a *System testing*. Every time that it is needed to test a component which is used by other ones that have not been integrated yet, an artificial environment is necessary for each integration test. The environment consists of driver programs and test data. It is also important to point out that in the system there are some external components, such as **Database** and **ExternalAPIs**, and so they are directly exploited without having to implement them, however they're integrated as soon as possible in order to test whether their interfaces can communicate properly.

The following diagrams illustrate the integration process at the various levels of dependency. (Note that it is not possible to apply a pure bottom-up approach due to the mutual dependency relationship between *ExternalAPIs* and some high level components).

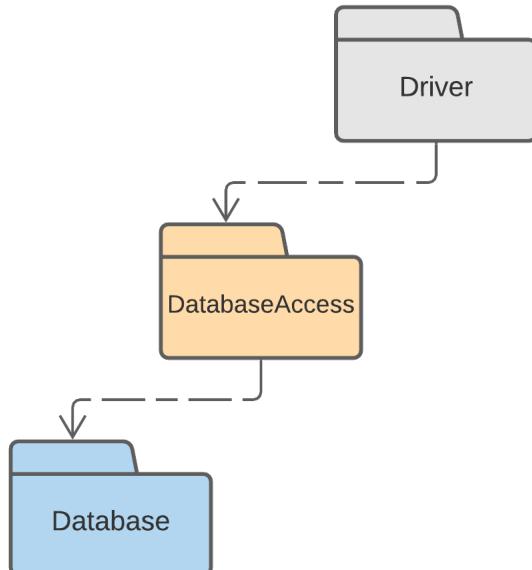


Figure 5.1: Integration of Database and DatabaseAccess components.

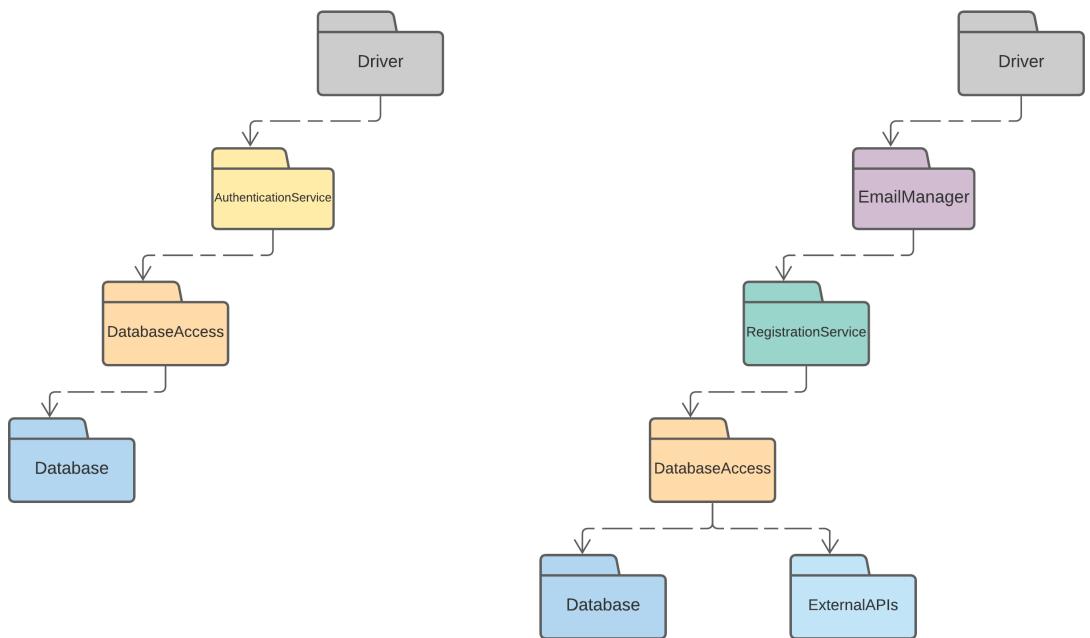


Figure 5.2: Login and registration services integration.

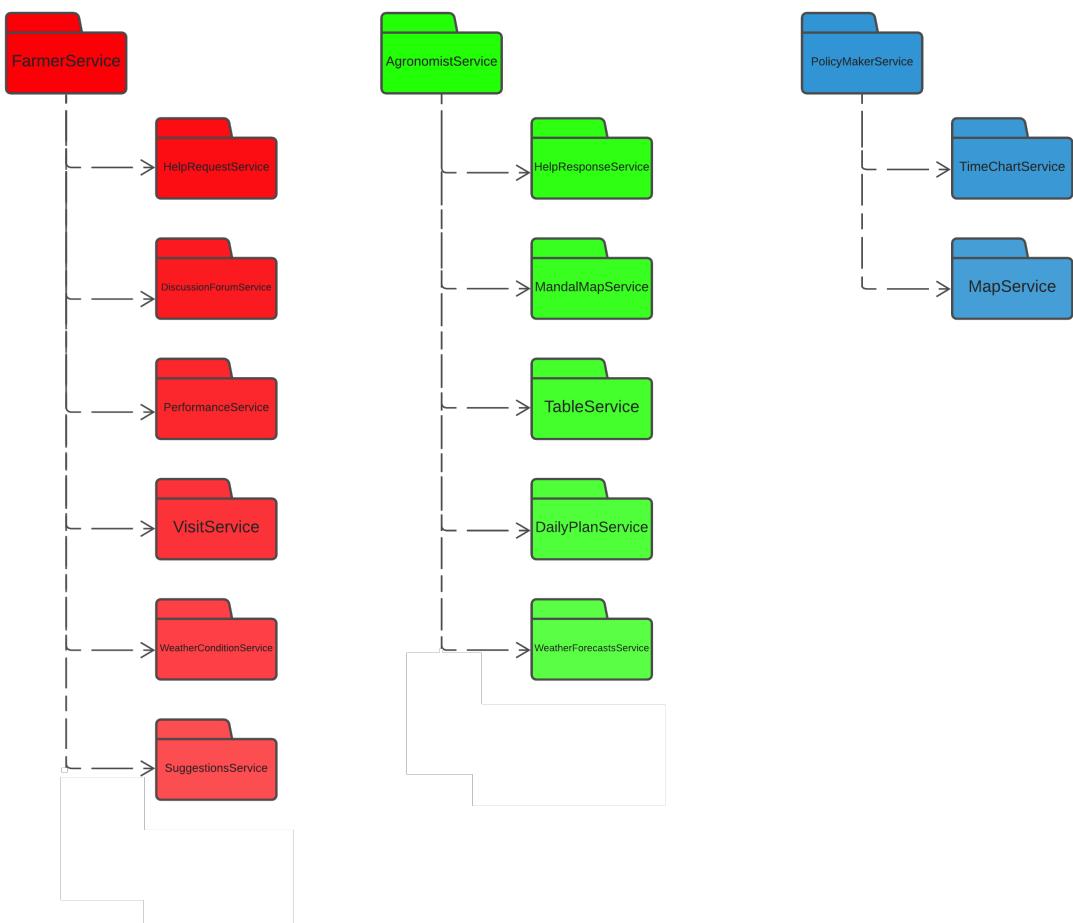


Figure 5.3: Integration of components for the three actors

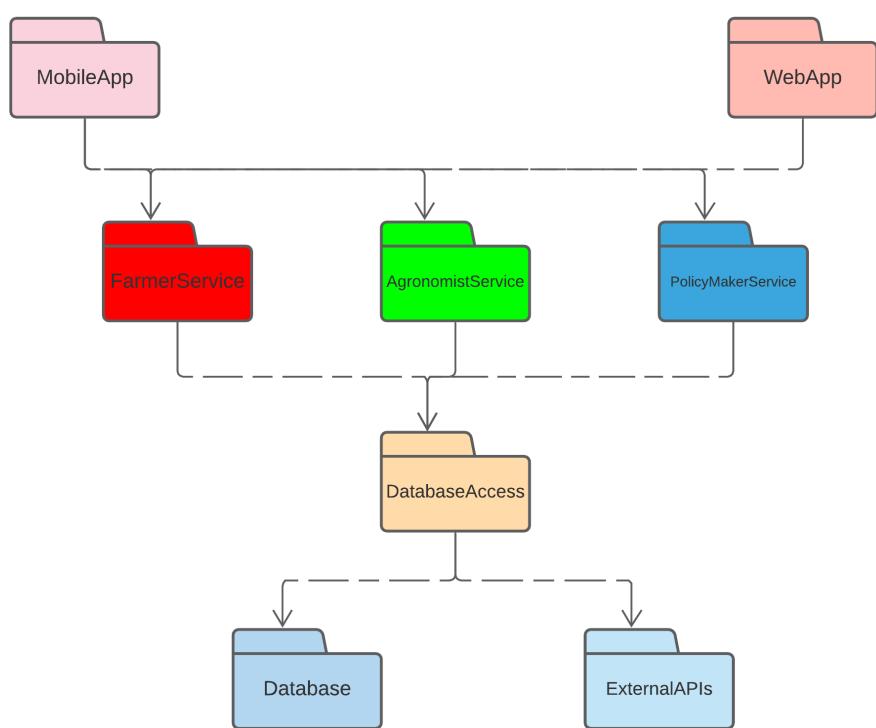


Figure 5.4: Integration of all main components

5.3 Testing Plan

Once all components have been integrated, programmers could perform *System Testing*. The aim of this kind of testing is to verify functional and non-functional requirements. Specifically, *DREAM* will be subjected to the following tests:

- **Functional Testing:** It ensures that the requirements and the specifications defined in the RASD are properly satisfied by the System.
- **Regression Testing:** This testing is done to make sure that new code changes should not have side effects on the existing functionalities. It ensures that the old code still works once the latest code changes are done.
- **Performance Testing:** This kind of testing identifies the performance bottlenecks in the system affecting response time, utilization, throughput.
- **Load Testing:** It aims at detecting bugs such as memory leaks, mismanagement of memory and buffer overflows. It also identifies the maximum operating capacity of the application.
- **Stress Testing:** It verifies stability and reliability of software application. The goal is measuring software on its robustness and error handling capabilities under extremely heavy load conditions and ensuring that software doesn't crash under crunch situations.
- **Database Testing:** It ensures data values and information received and stored into database are valid or not. Database testing helps to save data loss, saves aborted transaction data and no unauthorized access to the information.

Chapter 6

Effort Spent

6.0.1 Ali Arslan

Table 6.1: Effort spent - Ali Arslan

| Task | Hours |
|-----------------------|-------|
| Introduction | 1.00 |
| User Interface Design | 6.00 |
| Implementation Plan | 1.00 |
| Integration Strategy | 2.00 |
| Testing Plan | 2.30 |
| Document revision | 3.00 |

6.0.2 Servidio Elisa

Table 6.2: Effort spent - Servidio Elisa

| Task | Hours |
|--------------------------|-------|
| Overview | 1.30 |
| Runtime View | 8.00 |
| Component Interfaces | 2.00 |
| Other Design Decisions | 0.20 |
| Requirement Traceability | 1.30 |
| Document revision | 3.00 |

6.0.3 Suriano Federica

Table 6.3: Effort spent - Suriano Federica

| Task | Hours |
|----------------------------------|-------|
| Component View | 5.00 |
| Deployment View | 5.00 |
| Architectural Style and Patterns | 1.00 |
| Other Design Decisions | 0.20 |
| Requirement Traceability | 1.30 |
| Document revision | 3.00 |