POLITECNICO
MILANO 1863

Design Document

Authors: Pietro Valente
Andrea Seghetto
Professor: Damian Andrew Tamburri

Version: 1.0
Date: 10/1/2022
Repository GitHub: `https://github.com/pietrovalente/DREAM-software-engineering-2`

Contents

1 Introduction

1.1 Purpose

The purpose of this document is to provide more technical and detailed information about the system presented in the RASD document. It will represent a strong guide for the programmers that will develop the system in future.

Indeed, if the RASD has as its objective to provide a more abstract view of the system with its functionalities, the Design Document goes deeper into detail about the design, providing an overall guidance to the hardware and software architecture of the system. Here all the components forming part of the system and their interactions are described and all the design choices are listed and motivated. In general, the main different features listed in this document are:

- The high-level architecture of the system
- The main components of the system and their deployment
- The interfaces provided by the components
- The design patterns adopted
- Further details about the user interface
- A mapping of the requirements on the architecture's components
- Implementation, integration and testing plan

Stakeholders are invited to read this document in order to understand the features of the system being aware of the choices that have been made to offer all the functionalities satisfying the established functional and non-functional requirements.

1.2 Scope

We will deal with the design, the implementation and testing of DREAM, which is a data-driven predictive and also distributed system designed for governmental use and realized through the collaboration between the government of Telengana and some IT providers which must interface with multiple stakeholders. The main goal of the whole system is to monitor and optimize the farming production of the Indian state Telengana on which the food needs of a large part of the population depend, making it resilient to the many difficulties that threaten it, including phenomena related to climate change. As a distributed system we foresee the existence of a central server that include the front-end part, that is a user interface for webapp and application, with a simple design, to cope with the poor digital education of farmers and also back-end part that is a Database Management System (DBMS) which will take care of the storage and management of data. The system will use some external tools/APIs (i.e.TSDPS, Google Maps) to make available to its users some important features. One of these features lies in showing predictions about short or long-term climatic events in order to anticipate and govern them in the best possible way. We are considering three types of stakeholders: farmers, agronomists and policy makers. For each of these figures, the system provides some functions that will be described below.

As for policy maker the system guarantees them the ability to monitor the evolution of the entire national production process, distinguishing the farmers who have been more successful from those who needs help and allowing them to analyze whether the advice given by agronomists leads to significant results. To indentify the best farmers, an annual ranking of the farms registered is drawn up in real time until the deadline. In such a ranking each farm has its own score computed by the system following a precise formula and only the ten farms with the best score will be awarded; if a tie between them occurs all farms will be rewarded. Policy makers have the opportunity to see the ranking status with

the efficiency scores of the farms in real time and they reward the best farms only once a year, then the ranking is reset.

As for agronomists the system must assure them to write reports on the farms they visit containing the advice they have given with the help provided by DREAM AI; namely an AI algorithm of the system which proposes possible general suggestions on the production of each farm that the agronomist can choose to use or not. Only agronomists are allowed to give advice because they have the skills to do so, DREAM AI is just a tool used to help agronomists improving their advice which works by analyzing the evolution of the farm's production overtime and comparing it to the data related to other farms. The agronomists selected by the government are trained to give effective advice against climate change. Each agronomist is associated with an area, which corresponds to a province of Telengana.

Finally with regard to farmers, the system must allow them to view relevant information such as weather predictions and to request the intervention of the agronomist associated with them in case of in difficulty then they will take advantage of the personalized suggestions provided by him after the visit to the farm to improve production. Furthermore, it will be possible for each farmer to interface with others by asking them for help or by creating a new discussion on the related forum. The advice given in this area remains informal and its effectiveness is not monitored.

The few paragraphs just read represent an overview of the main functionalities offered by the system: more detailed information can be found on the RASD document.

1.3 Definitions, Acronyms

1.3.1 Definitions

Validation code	These codes are used to identify users for who they really are. Each user will receive an email invitation to register on the platform, with a personal code for registration. Validation verifies the correctness of the code.
Validation data	Each time data is entered by the user, a validation process verifies that the characters do not exceed the maximum number and that there are no harmful strings (i.e. SQL injection).
Query	Request made to the system in order to find information.
Webapp	Application software that runs on a web server.
Zone	A province of Telengana to which an agronomist is associated.

1.3.2 Acronyms

RASD	Requirements Analysis and Specification Document
DREAM	Data-dRiven PrEdictive FArMing in Telengana
DREAM AI	Data-dRiven PrEdictive FArMing in Telengana Artificial intelligence
TSDPS	Telangana State Development Planning Society
API	Application Programming Interface
DD	Design Document

HTTP	HyperText Transfer Protocol
IP	Internet Protocol
SQL	Structured Query Language
JSON	JavaScript Object Notation
FSM	Farm Score Matrix

1.4 Revision history

For the history of the creation and modification of the document refer to the link: `https://github.com/pietrovalente/ValenteSeghetto/commits/main`

Date	Modifications
6/1	First final version, is missing: • Run-time view: 6.New report is inserted description • Other design decisions
7/1	Second final version • Introduced last Run-time view: 6.New report is inserted description • Some corrections in the section Run-time view • Missing: Other design decisions
9/1	Third final version • Introduced "Other design decisions" section • Some corrections in all document • Add the effort time
10/1	Fourth final version • Forgotten to add MVC pattern description to final version

1.5 Document Structure

- **Chapter 1** describes the scope and purpose of the DD, including the structure of the document and the set of definitions, acronyms and abbreviations used.
- **Chapter 2** aims to provide a description of the architecture design of the system, it is the core section of the document. More precisely, this section is divided in the following parts:
  - Overview: High-level components and their interaction
  - Component view
  - Deployment view
  - Runtime view
  - Component interfaces
  - Selected architectural styles and patterns
  - Other design decision
- **Chapter 3** shows how the user interface should be on the mobile and web application.
- **Chapter 4** describes the connection between the RASD and the DD, showing the matching between the goals and requirements described previously with the elements which compose the architecture of the system.
- **Chapter 5** includes the description of the implementation plan, the integration plan and the testing plan, specifying how all these phases are thought to be executed.
- **Chapter 6** shows the effort spent for each member of the group. Finally, there is a section dedicated to all the references used.

2 Architectural design

2.1 Overview

DREAM is designed to be structured as a distributed system, whose architecture is organized in three logic layers:

- **Presentation level (P)** this level deals with user interaction and therefore contains all the interfaces that allow him to communicate easily with the system, requiring its functionalities.
- **Business logic or Application layer (A)** this level incorporates the business logic of the system and the functions it offers to users; moreover, placing itself halfway between the other two layers, it is responsible for moving data between them and in general for coordinating the work of the entire system.
- **Data access layer (D)** this level is responsible for managing the information in the system, in particular its storage and retrieval through corresponding operations and access to the databases

Another architectural principle followed in the design of the system is the client-server style. Specifically, client and server are located on separate physical machines and communicate through interfaces and system components, each interaction involves a request for a certain functionality moved by the client that is taken over by the server and fulfilled through the invocation of appropriate methods /routine.

Some of the main functions offered by the system are listed below, divided according to their degree of complexity in basic services or advanced functions if they require more processing.

Basic Service:

- A farmer requests to update his data: the server receives the update request with the data to be entered; therefore, it acquits it by storing the data in the database after verifying its validity.
- A farmer requires the intervention of an agronomist: the server receives the request and after verifying the validity of the data reported by the client with the request, stores them in the database. Once the request has been stored, the task of alerting the agronomist will then be up to the component of the server delegated to notify.
- A farmer requests to view a discussion in the forum: the server receives the request, retrieves the desired information from the database and sends data back to the client.
- A farmer requests to create or comment a discussion on the forum: the server receives the request with the data the client wants to enter; therefore, it acquits it by storing the data in the database after verifying its validity.
- An agronomist requests to upload a report: the server receives the request for the data to be entered; therefore, it acquits it by storing the data in the database after verifying its validity.
- An authorized user requests to view the data of a farm or the ranking of the farms at that moment: the server receives the request, retrieves the desired information from the database and sends data back to the client.

Advanced Functions:

- The system proposes to agronomists the advice processed by its component related to DREAM AI: the server processes the advice through its component and stores the data in the database, subsequently when the client associated with an agronomist request to view the data of a certain farm, the server also retrieves the processed recommendations from the database and sends data back to the client

- A policy maker wants to evaluate the work of an agronomist: the server receives a request from the client associated with the policy maker to view the state map; the server fulfills the request by interfacing with Google Maps. After selecting an area in the map, the client requests to view the data and reports of the farms in that zone to evaluate the effectiveness of the advice given from time to time by the competent agronomist; the server fulfills the request by retrieving information from the database and sending data back to the client.

The three logic layers described above are divided into as many hardware layers called tiers, each associated with a logic layer and made up of a machine or a group of machines, such that each logic layer has its own dedicated hardware in the system. The hardware architecture chosen is therefore Three-Tier, where a tier is dedicated to the interface with the user, a middle tier is associated with the application level and a last one is reserved for the database management server. As mentioned, this type of architecture introduces a second tier containing the business logic of the application to physically separate clients and data, thus significantly improving system security because the control of database access by users is always mediated by the intermediate tier.

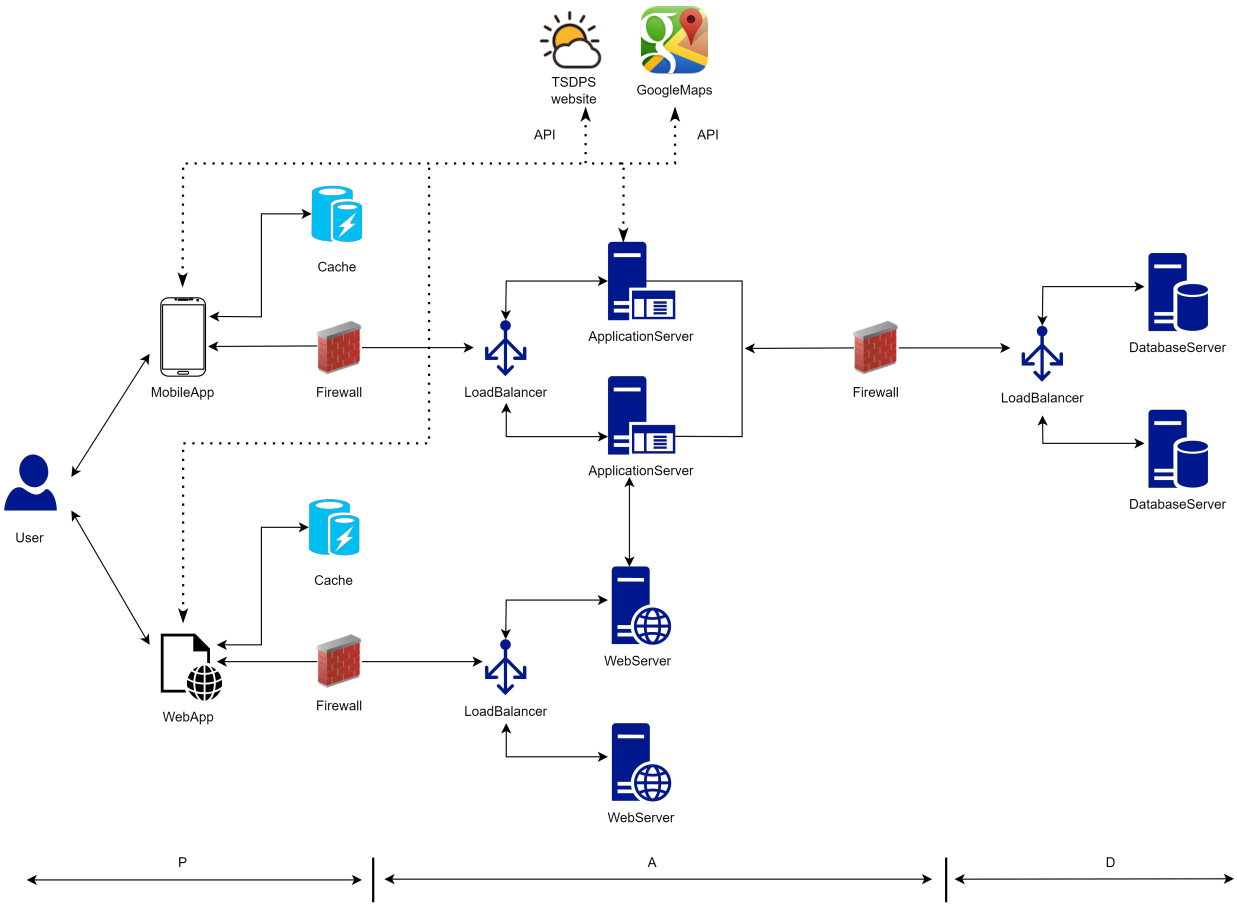


Figure 1: System Architecture

As shown in the figure above, the client's devices that can be used to access and interface with the system can be computers or mobile phones. Smartphones using the DREAM mobile app connect directly to the application server, while personal computers using the web app connect to the system's web server. The servers communicate with each other, specifically the web server interfaces with the application server to guarantee clients connected via PC the use of the same features offered by the mobile app. The application server unlike the web server is able to interface with the Google Maps API external to the system and use it to allow clients to view the map of the state, moreover it is also capable of responding to requests for viewing the weather forecast by redirecting the client to the corresponding TSDPS website. Finally, the application server also has the task of interacting with the database server (DBMS) to store information or retrieve it.

In order to guarantee and improve one of the main properties required from the system, namely scalability, it was decided to use the technique of node replication. Observing the figure above it is in fact possible to notice how both the web server and the application server are available in the system in two copies that share all the requests coming from the clients through the action of specially set up load balancers. The same technique is also applied for the database server with an additional load balancer used to distribute the computational load between the two available nodes. In general, in addition to improving the scalability of the system, replication also contributes to improving its security through data replication, making the system fault tolerant.

To fasten and lighten the communication, caches are used in front of the Presentation tier: the cache needs to have an appropriate knowledge of data that can store, this knowledge is required to invalidate data when needed (better explained in Section 2.6).

At the time of connection to the site, through the cache mechanism, the users' cache stores some important and expected to be seen information, like part of the users' data and stats (for a farmer data farm information); so that they do not even have to be connected to the Internet to consult them. This can help a lot in cases where the internet connection is not very good.

Finally, concerning the security of the system, it is important to emphasize that the proposed hardware architecture provides for the installation before and after the application tier of firewalls that prevent the data stored in the system from unwanted access by external parties. This protection measure is important given the large amount of sensitive information handled by the system.

2.2 Component view

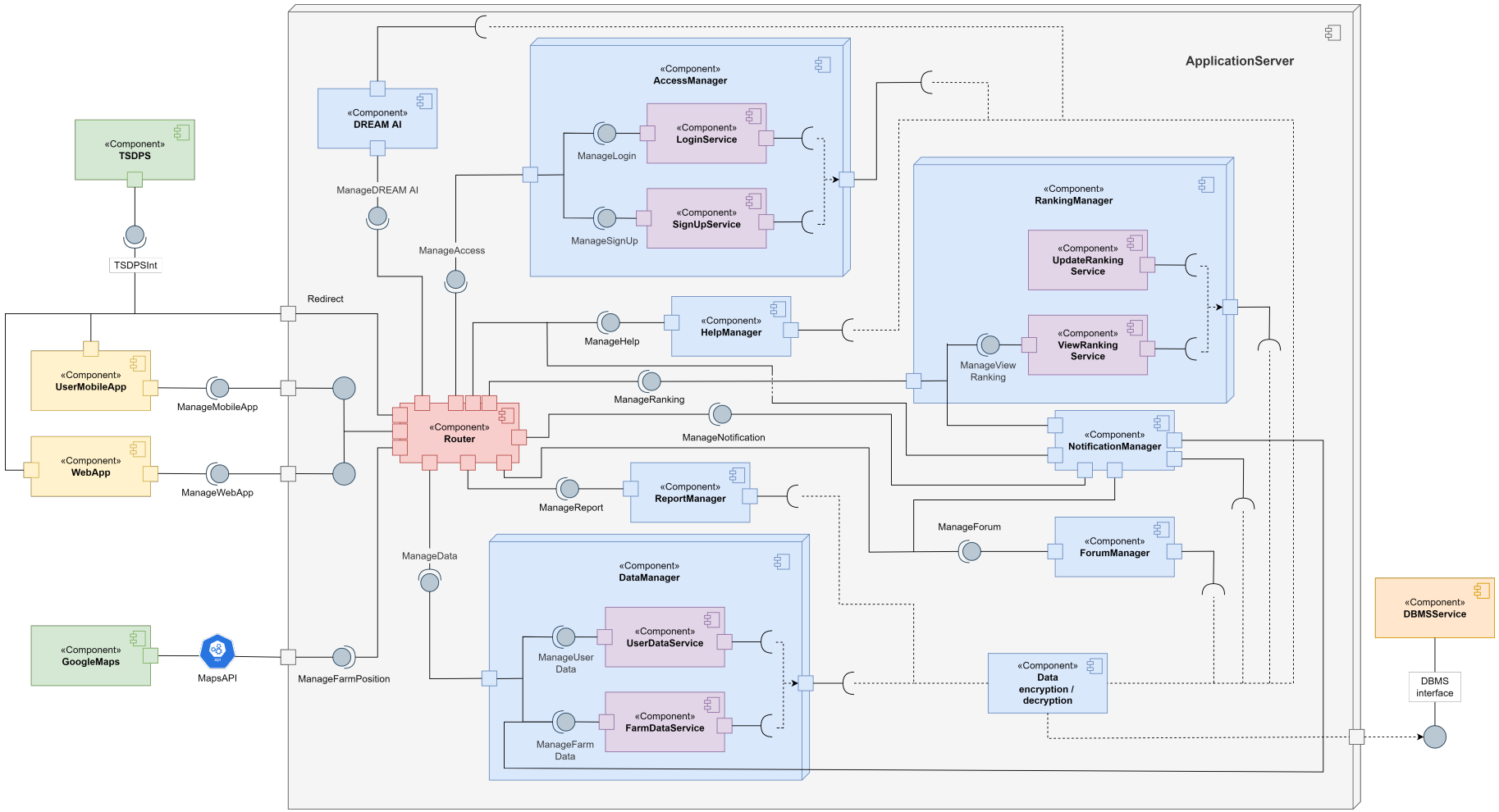


Figure 2: Component Diagram

The previous component diagram gives a specific view of the system focusing on the representation of the internal structure of the application server, showing how its components interact each other.

- **UserMobileApp and WebApp:** these are the user-side access points (see Section 2.3 for better explanation).

- **Router:** it provides two different interfaces to UserMobileApp and WebApp, handling all requests coming from them. In this way, every request is handled by this component, which is the unified manager of the whole system (Façade pattern, see Section 2.6.2).

  Another very important function of the router is to allow interaction between the components of the server, in fact these communicate with each other only through it (Mediator pattern, see Section 2.6.3). For example, before execute each client request, the router will automatically check the credentials, using the LoginService (and using information from the referring client token). Logically this is true when referring to functions, because for data usage each component connects independently to the DBMS (through Data encryption / decryption component). This modular structure of the components allows for greater organization and clarity.

  The router also takes care of redirecting the user to the weather site (TSDPS) and interfacing with the external google maps API.

- **AccessManager:** this component manages access to the system and comprises two subparts

  - **LoginService:** manages the access requests to the system by checking the credentials. If these are present in the database, at the first access, it returns a token (together with the response) which is saved on the client side and contains the user's status. This will allow the user not to have to re-enter the request every time he makes a new one.
  - **SignUpService:** manages the registrations of new users in the system. In particular, check that the user/farm code is present among the system codes, this will only allow users with the requirements to enroll in the system. Once used for a registration, the code will be invalidated.

- **DREAM AI:** manages the implementation of the algorithm that helps agronomists to give advice to farms (described in Section 2.8).

- **RankingManager:** this component manages the ranking and comprises two subparts

  - **UpdateRankingService:** this service takes care of keeping the ranking updated once a minute, the ranking changes in case a new farm registers or every time a farm updates its production data. Given that updates are expected to be from many farms, and the cost of updating is onerous, it was decided to use such a solution instead of one with the Observer pattern (ranking update every time it occurs a change). It is a component that works disconnected from the rest of the server, even if it is part of it, this is the reason why it is not connected to the router.
  - **ViewRankingService:** this component takes care of displaying the ranking. This is updated every time you reload the page or return to the appropriate section of the app.

- **ForumManager:** this manager takes care of the forum, both viewing and creating (saving in database) discussions and comments. It allows the interaction between the farmers, because only they can access this functionality.

- **ReportManager:** manages the creation, visualization and saving of reports written by agronomists. This component provides the possibility to interact with DREAM AI,Google Maps, TSDPS and DataFarmService (via Router), to allow agronomists to have all the information necessary to give the right advice to farms.
- **NotificationManager:** the functionalities of this component are deeply rooted in the other components, this explains why it is the only component that bypasses the unified management of the interaction between the components provided by the router. The main notifications concern the forum (tags in comments and comments on discussions), the ranking (when the final one is ready and the best ones are asked to tell their experience), a ask for help from a farmer and the farm data (notify if they have not been updated or if the consumption of water is above the threshold).

  Based on the type of notification, the related service operates differently, for notifications that have a time or day, the relative checks will be carried out in those moments (for example the final ranking or the data update at 21 if the farmer has not updated the data), otherwise the service will carry out periodic checks of the data (for example 1 per minute for the case of notifications relating to the forum or 1 per day for the threshold of water consumed in the farm).
- **HelpManager:** to request this feature, the farmer must fill out a forum specifying the problems he is facing and for what reasons he requires help, this component has the task of saving this document in the database and proposing it to the agronomist, forcing him to manage the request for help.
- **Data encryption/decryption:** this component allows to encrypt the components of the queries to be made to the database, or to decrypt the data from the database. This happens for security reasons (a data leak in the database, without being able to decrypt it would not be too catastrophic).
- **DataManager:** this component takes care of data and comprises two subparts
  - **UserDataService:** manages the storage, modification and display of user data.
  - **FarmDataService:** manages the storage, modification and display of farm data.
- **DBMSService:** this is the component that allows every other component in the system to interact with the database. The Interface provided by this component contains all useful methods to store, retrieve, update data into the database from different actors. Every internal component of the application server uses some methods of its interface.

- **TSDPS:** this component is external to the server and indicates the weather site. The router can only redirect the user to the site.
- **GoogleMaps:** it is used for a graphical display of farms on the map. This component uses APIs to make google maps interact with the farms present in the system.

2.3 Deployment view

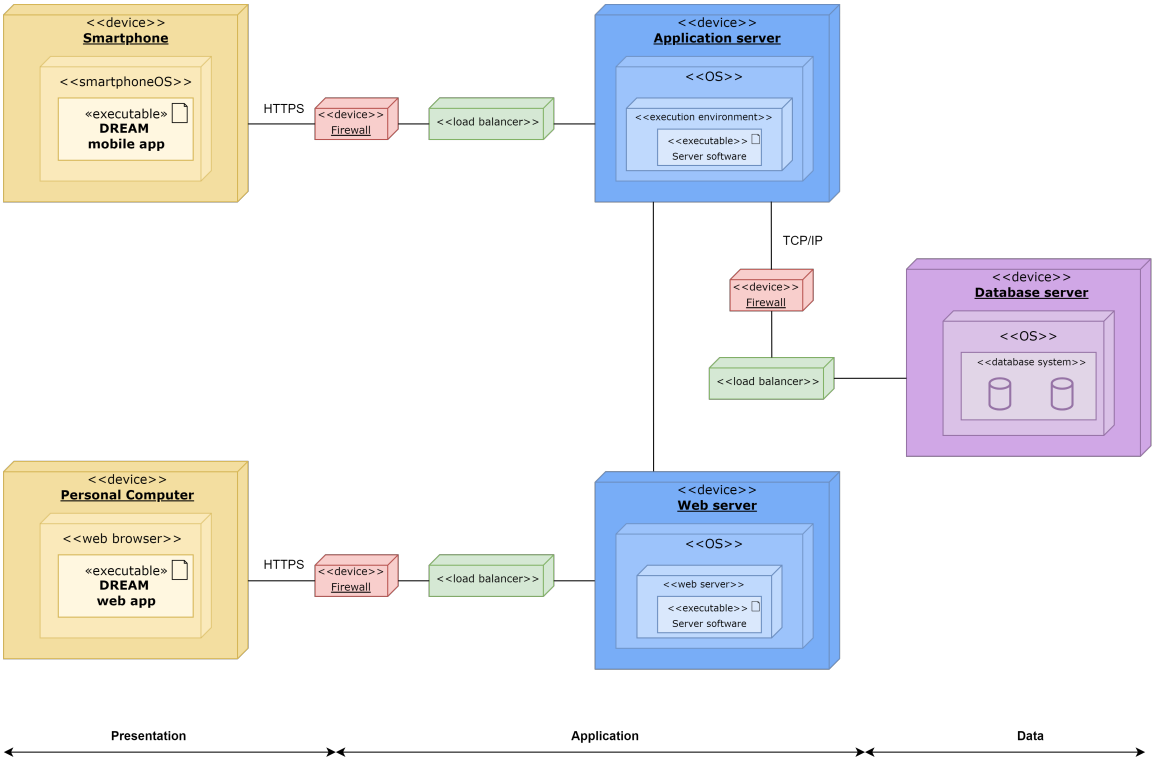


Figure 3: Deployment Diagram

As mentioned, the system architecture is organized into three tiers which contain:

- **Tier 1:** here is the presentation logic of the system. The main elements that we find in this tier are the devices necessary for users to access and use the system; specifically, it is possible to use the DREAM mobile application for smartphones or access from your personal computer via the webapp. The mobile application allows smartphones to connect directly to the application server via HTTPS, thus being able to take advantage of all the features offered by the system quickly and easily even in conditions of mobility. In addition, the mobile application must be available for both Android and iOS for it to be usable by most of the devices in circulation. The web application, on the other hand, is designed for those who do not have a smartphone or do not have an operating system compatible with the mobile application. This second method of accessing DREAM allows the client to connect via HTTPS to a web server capable of performing most of the services offered by the system; the features not guaranteed by the web server will in any case be made available to web users through its communication with the application server. The desktop version of DREAM is supported by Windows, Linux and Mac operating systems.

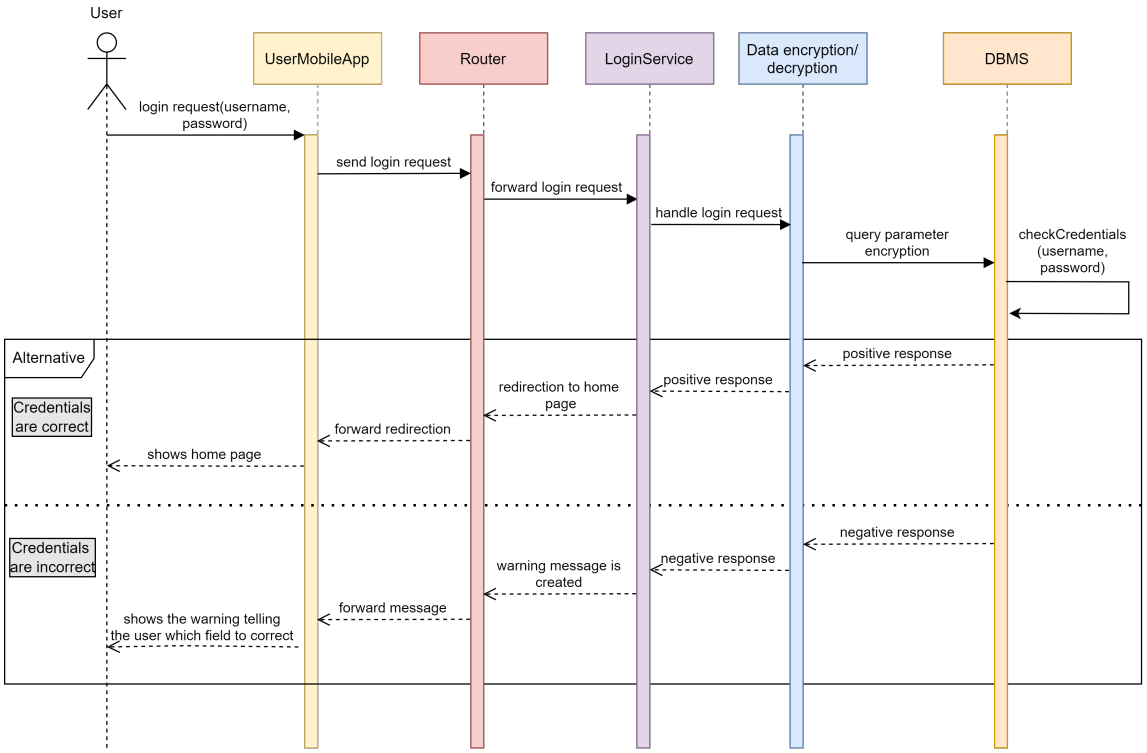
- **Tier 2:** here is the application logic of the system. The application server manages all the requests it receives from clients and provides the appropriate responses thus ensuring all the various services offered by DREAM. The web server, on the other hand, satisfies only some of the requests coming from the clients, for others such as the redirection of the user to the TSDPS site or the management of the Google Maps API external to the system, the web server relies on the application server to make these features equally available to its users. Furthermore, even in accessing and managing the database, the web server relies on the application server, because it is the only one capable of communicating via TCP / IP with the DBMS. This tier also includes the load balancers shown in the figure above, necessary for the replication of the web server, application server and database

server nodes chosen for the system design. Specifically, the load balancers at the server level are useful for sharing the system workload among multiple nodes, increasing both the capacity and the reliability of applications and avoiding the overload of any single server. Finally, it should be noted that immediately before and after the servers there are firewalls that guarantee secure access to the internal network of the system and prevent any unwanted external access / attack.

- **Tier 3:** here is the data logic of the system. The database server is meant to run a relational DBMS. The choice of having a relational database is due to the possibility that this offers to connect information from different tables using foreign keys. Furthermore, a database of this type lends itself well to managing complex queries and database transactions. The replication of the nodes, managed also in this case through the use of a load balancer in charge, allows the replication of data in the system and is therefore important to ensure the safety and tolerance to faults of the system.

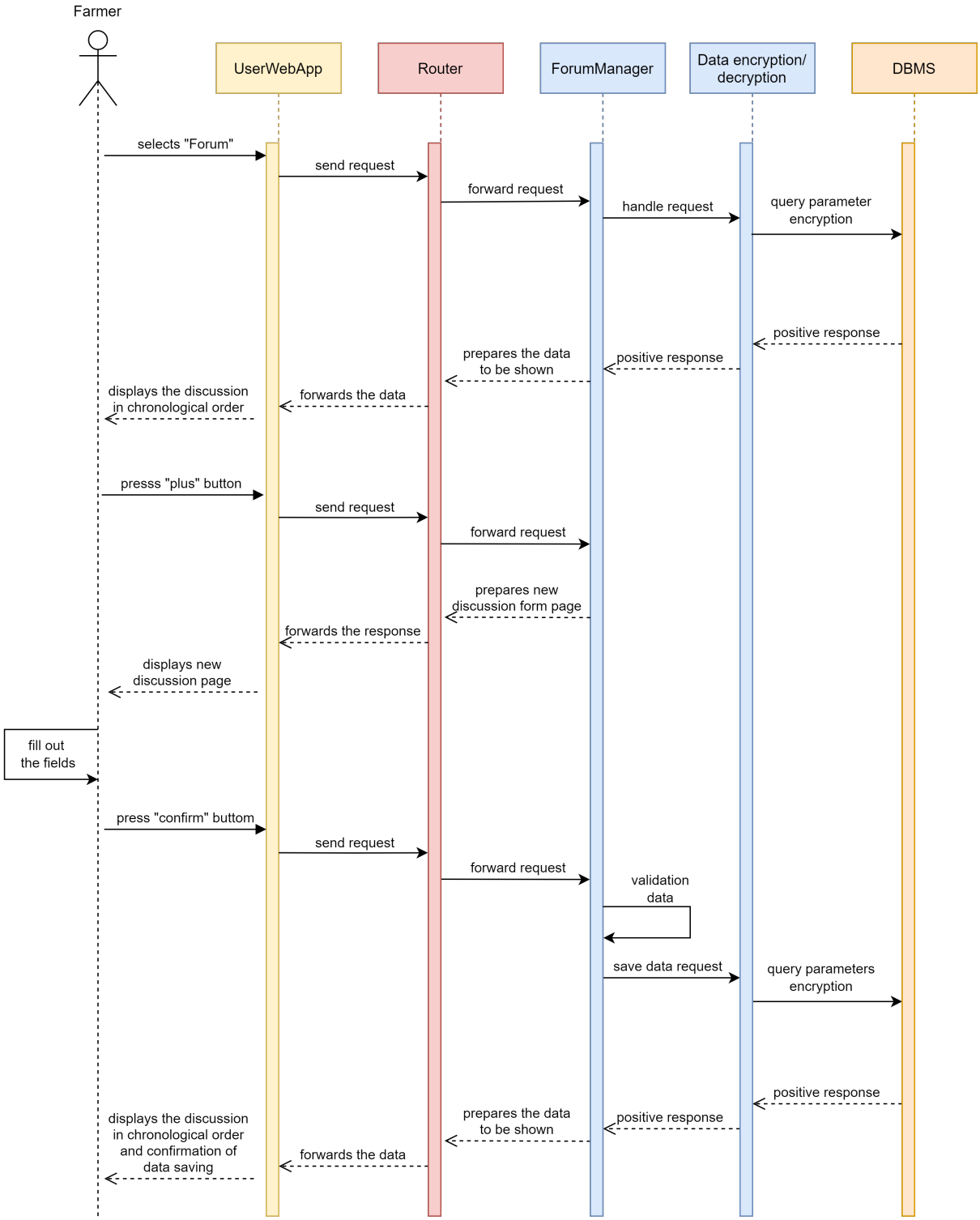
2.4 Run-time view

1.User logs in



This sequence diagram shows the login process inside the system that the users request goes through. The login request is made by the user by filling in the appropriate form with username and password through the mobile app or from a computer through the web app, depending on the chosen access method. Subsequently, the request passes to the router of the application server or web server which then forwards it to the LoginService, that is the component of the server delegated to manage login requests. The latter then interfaces with the server component reserved for data encryption / decryption, specifically the login request is translated into appropriate queries to be performed on the database and their parameters are encrypted. Finally, the credentials are verified on the database through the invocation of checkCredentials with which it is possible to check if the username entered by the user in the system exists and if the password matches. The result of the operation is propagated back to the user and if the result is positive the login will be successful, otherwise a warning message indicating which field to correct is shown and the request will have to be repeated.

2.Start a discussion

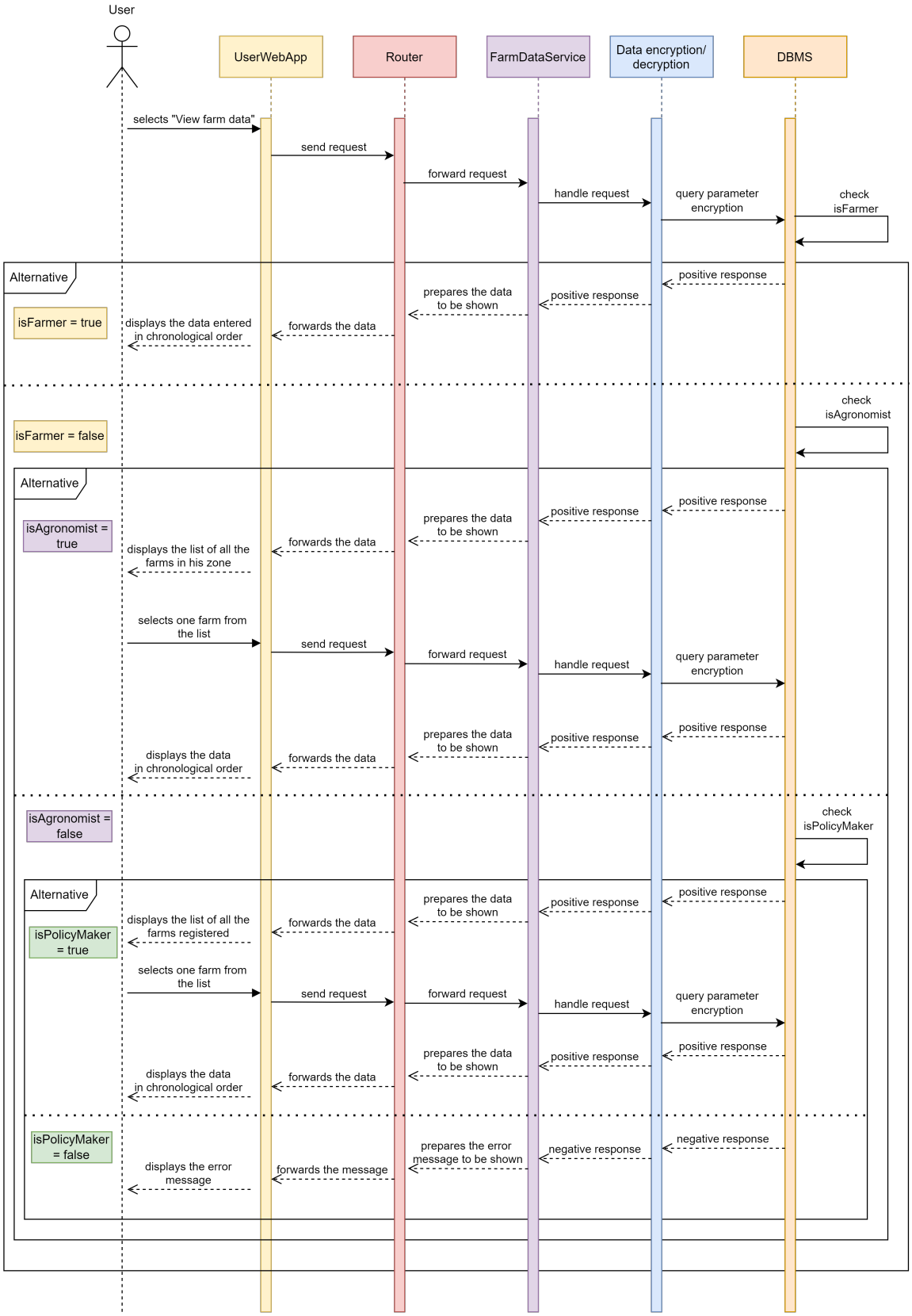


This sequence diagram shows the process inside the system that the farmer interacts with the forum, in particular in creating a new discussion. It is assumed that the farmer making the request is already logged into the system and is therefore shown the features specific to his role, among which the farmer chooses precisely "Forum" option. Subsequently, the request passes to the router of the app/web app. At this point, the ForumManager has the task of retrieving all the most recent discussions from the database, to do this the request must pass through the Data encryption / decryption component that must decrypt some data of the database tables (sensitive data). Once the discussions are obtained the data is sent to the router to be prepared to be displayed. Subsequently the router provides an interface of the page with the data to be shown to the app/web app, which uses the response of the server to organize the data dynamically in the page to be shown to the user.

If the user wants to add a new discussion, he will press the "plus" button, generating a request that the router forward to ForumManager. In response, this component will send the information it wants to receive from the user to the Router (form fileds), the Router together with the app/web app will show this information to the user. Once the discussion page form has been filled in, the user pressing "confirm" will trigger a set of requests that will retrace the components mentioned above, until the new data is saved in the database.

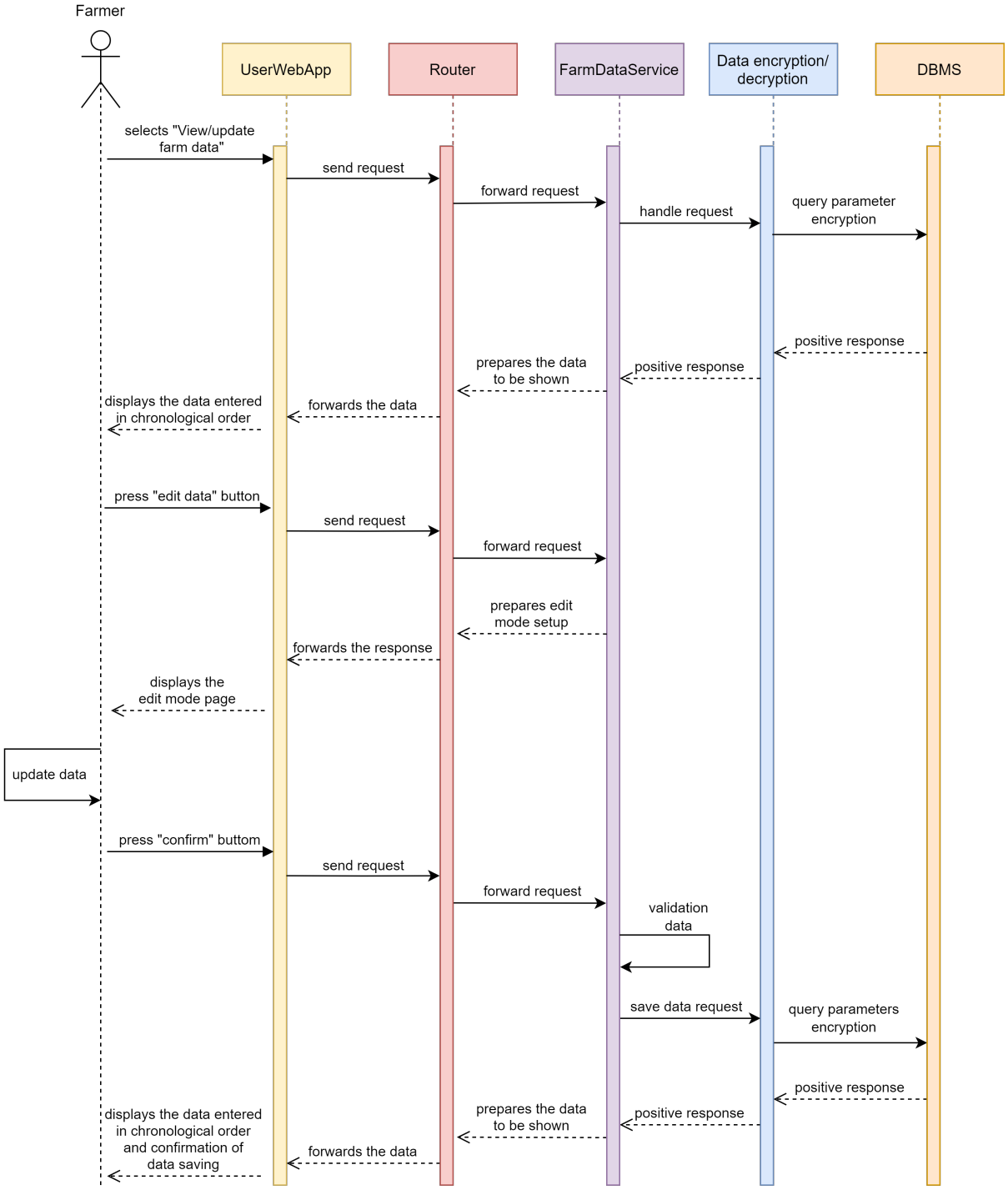
An important step concerns the validation data (carried out by ForumManager) for checking the data entered, for example description no longer than that stability, presence of special characters not accepted by the chosen encoding, etc.

3.View farm data



This sequence diagram shows the process inside the system that the users request for viewing farm data goes through. It is assumed that the user making the request is already logged into the system and is therefore shown the features specific to his role, among which the user chooses precisely view farm data. Subsequently, the request passes to the router of the app/web app, which then forwards it to the FarmDataService, that is the component of the server delegated both to update and view the farm data. The latter then interfaces with the server component reserved for data encryption / decryption, specifically the view request is translated into appropriate queries to be performed on the database and their parameters are encrypted. Finally, the user identity is verified on the database through the invocation of some methods that are isFarmer, isAgronomist and isPolicyMaker, this check is necessary because the response to the request, as well as the data shown depend on the user category. Specifically, if the user is a farmer, then the positive response to isFarmer is propagated up to the FarmDataService which prepares as data to be displayed those of the associated farm, the data will then pass through the router and will finally be forwarded to the mobile/web, which allows the farmer to view the data entered in chronological order. If the requesting user is not a farmer, then if isFarmer gives a negative answer, it is checked whether the user is an agronomist or not with isAgronomist; in the affirmative the propagation of the response follows the same path described above for isFarmer up to FarmDataService. The latter prepares the display of the list of all the farms in the area of competence of the agronomist and passes the data to the router which forwards them to the app/web app, the agronomist then ends up displaying this list of farms from which he can choose one in specific to examine. If isAgronomist also provides a negative answer, then a final check is made on the user's identity, checking whether the user is a policy maker or not with isPolicyMaker; in the affirmative, the answer follows the same path described in the other cases up to the FarmDataService which in this case prepares the display of the list of all the farms registered in the system. The prepared data is once again passed to the router, then forwarded to the app/web app and finally shown to the policy maker who will have then the possibility to choose any farm from the list to examine. If all the identity checks performed fail, namely in case isFarmer, isAgronomist and isPolicyMaker all give a negative response, then obviously an error has occurred. The error is managed by the FarmDataService through the creation of a message for the user. Finally, the message is passed to the router and from this forwarded to the app/web app to be shown to the user.

4.Update data

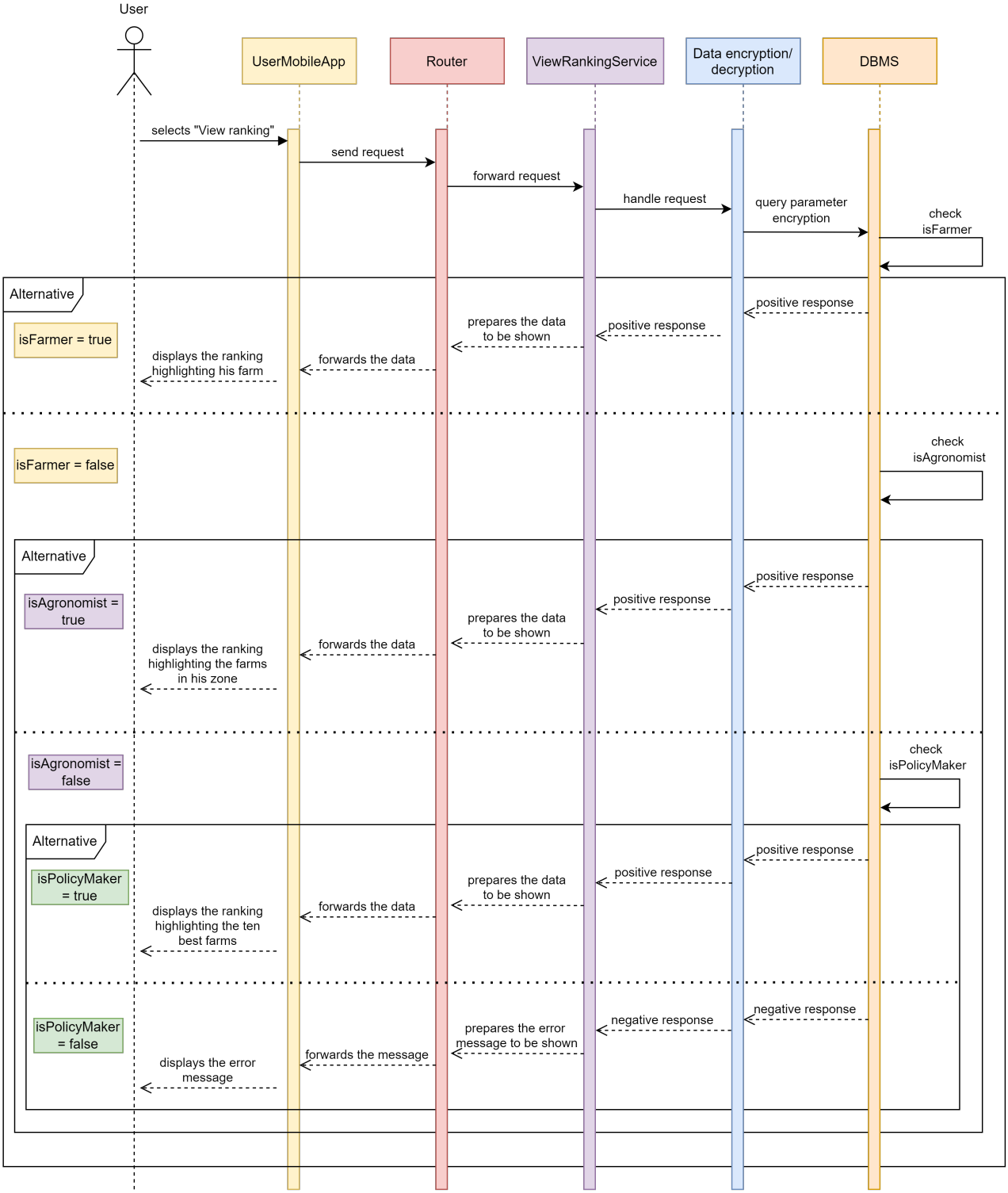


This sequence diagram shows the process inside the system that the farmers request for updating farm data goes through. This process is different from the others in that it is not possible to access this function directly from the menu. It is assumed that the farmer making the request is already logged into the system and to be able to update the farm data he must first go through the data visualization functionality, described in the previous run-time view. For this reason, unlike the "View farm data" functionality present for the agronomist and policy maker, for the farmer this is called "View / update farm data".

Once the farmer is in his farm data view screen, he can press the "edit data" button. At this point the app/web app sends the request to the server which foward the request to the FarmDataService. In response, this component will send the information concerning the data that the farmer can modify, the Router together with the app/web app will show this information to the user, letting it enter "edit mode".

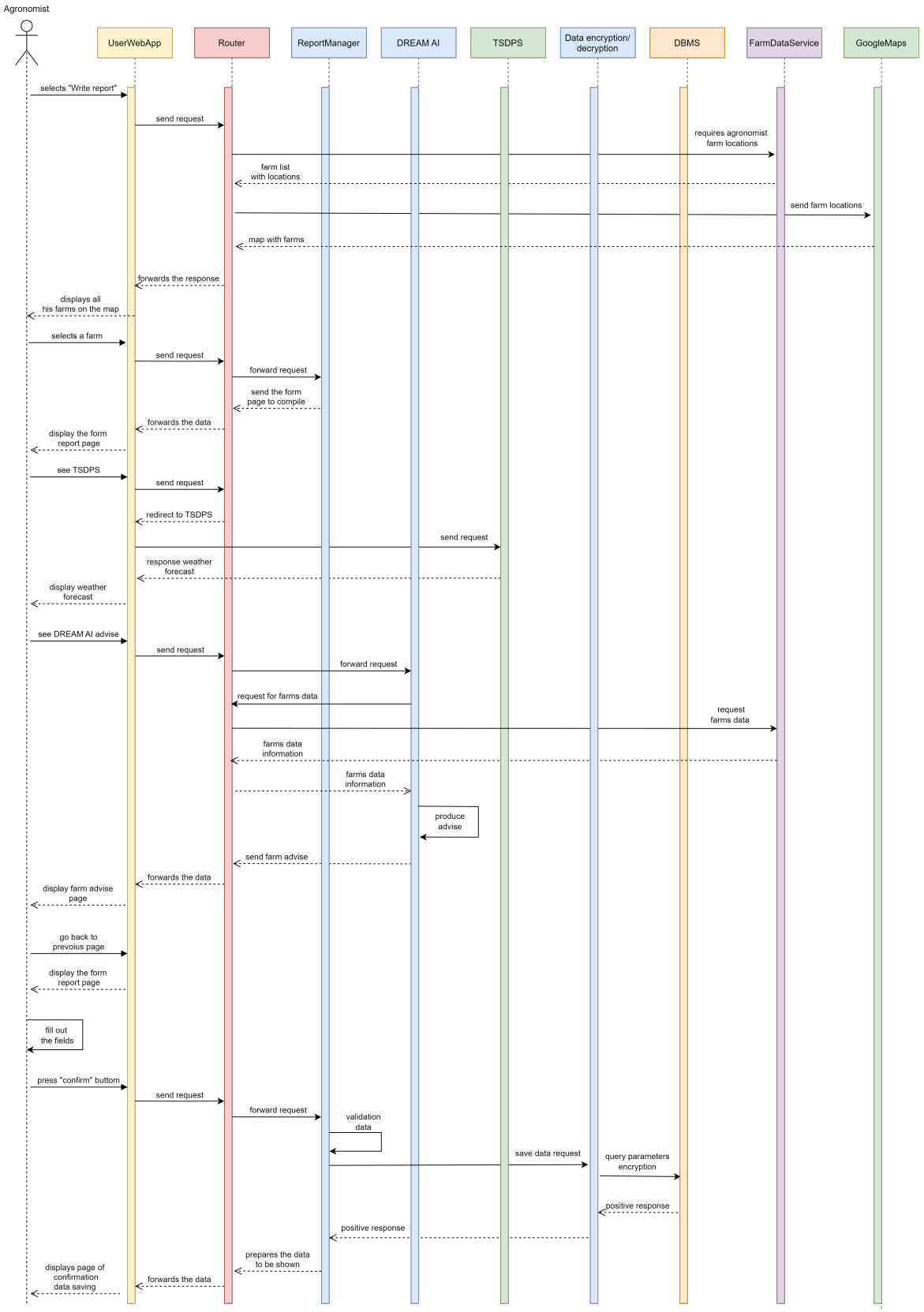
When the user has changed the data, setting them to the daily ones, pressing the "confirm" button will start a chain of operations for saving the data. From the app/web app the request is handled by the Router, which in turn redirects it to the FarmDataService. From here the request is validated and subsequently sent to the Data encryption / decryption component, to then save the data in the database. If everything happens correctly, a response cascade is generated, up to the app/web app, displaying the updated farm data.

5.User views the ranking



This sequence diagram shows the process inside the system that the users request for viewing ranking goes through. It is assumed that the user making the request is already logged into the system and is therefore shown the features specific to his role, among which the user chooses precisely view ranking. Subsequently, the request passes to the router of the application server or web server, depending on the chosen access method, which then forwards it to the ViewRankingService, that is the component of the server delegated to acquit this type of requests. The latter then interfaces with the server component reserved for data encryption / decryption, specifically the view request is translated into appropriate queries to be performed on the database and their parameters are encrypted. Finally, the user identity is verified on the database through the invocation of some methods that are isFarmer, isAgronomist and isPolicyMaker, this check is necessary because the response to the request depend on the user category. Specifically, if the user is a farmer, then the positive response to isFarmer is propagated up to the ViewRankingService which prepares the display of the ranking highlighting the item that corresponds to the farmer's farm, the data will then pass through the router and will finally be forwarded to the app/web app to be shown. If the requesting user is not a farmer, then if isFarmer gives a negative answer, it is checked whether the user is an agronomist or not with isAgronomist; in the affirmative the propagation of the response follows the same path described above for isFarmer up to ViewRankingService. The latter prepares the display of the ranking highlighting the items corresponding to the farms in the area of the agronomist and passes the data to the router which forwards them to the app/web app and thus the ranking is finally displayed. If isAgronomist also provides a negative answer, then a final check is made on the user's identity, checking whether the user is a policy maker or not with isPolicyMaker; in the affirmative, the answer follows the same path described in the other cases up to the ViewRankingService which in this case prepares the display of the ranking highlighting the items corresponding to the top ten best farms. The prepared data is once again passed to the router, then forwarded to the app/web app and finally shown to the policy maker. If all the identity checks performed fail, namely in case isFarmer, isAgronomist and isPolicyMaker all give a negative response, then obviously an error has occurred. The error is managed by the ViewRankingService through the creation of a message for the user. Finally, the message is passed to the router and from this forwarded to the app/web app to be shown to the user.

6.New report is inserted



This sequence diagram shows the process inside the system that the agronomist request for writing a report goes through. It is assumed that the agronomist making the request is already logged into the system and is therefore shown the features specific to his role, among which the user chooses precisely "Write report".

When the request arrives at the Router, the first step involves a request to FarmDataService for the address of all the farms under the responsibility of the agronomist who is making the request. The dictionary with this information that FarmDataService returns as a response is used by the Router to interface with GoogleMaps, which responds with a map with all the reported locations. This information is sent to the app/web app with some information on how to graphically display the page with this external component.

At this point the agronomist can select a farm from the map, the request is bounced to the Router which interacts with ReportManager in order to obtain the necessary information for a report (form fields). The router will send this information to the app/web app along with the information for the pagination / layout.

Proceeding the agronomist can interact with the report writing page, which in addition to the possibility of compiling and sending information, also offers some important functions, which are: "see production data", "see DREAM AI advise", "see TSDPS ".

If the user needs to view the farm data, he can do it and the system will respond as shown in the diagram above (3.View farm data).

If the user wants to see the weather forecast, he can press the relative button, as shown in the diagram, the request will arrive at the router which will have the task of redirecting, by opening a new page, the app/web app on the TSDPS official website. At this point the interaction between the user and the site is no longer managed by the DREAM system.

Returning to the report page, if the agronomist presses on "see DREAM AI advise", the request, once it reaches the Router, triggers a series of cascading operations. The Router asks the DREAM AI component for advice for the farm in question, this component, in order to give an answer, must first ask the Router to obtain the farm data via FarmDataService. Once the data is obtained, DREAM AI processes them as explained in Section 2.8, producing the advises. These advises are then sent to the Router which forwards them to the app/web app on a new page.

Once the data has been consulted, the agronomist can return to the previous screen and proceed with the compilation of the report.

When the agronomist has entered the information, pressing the "confirm" button will start a chain of operations for saving the data. From the app/web app the request is handled by the Router, which in turn redirects it to the ReportManager. From here the request is validated and subsequently sent to the Data encryption / decryption component, to then save the data in the database.

If everything happens correctly, a response cascade is generated, up to the app/web app, displaying the homepage and a message indicating the correct insertion of the report.

2.5 Component interfaces

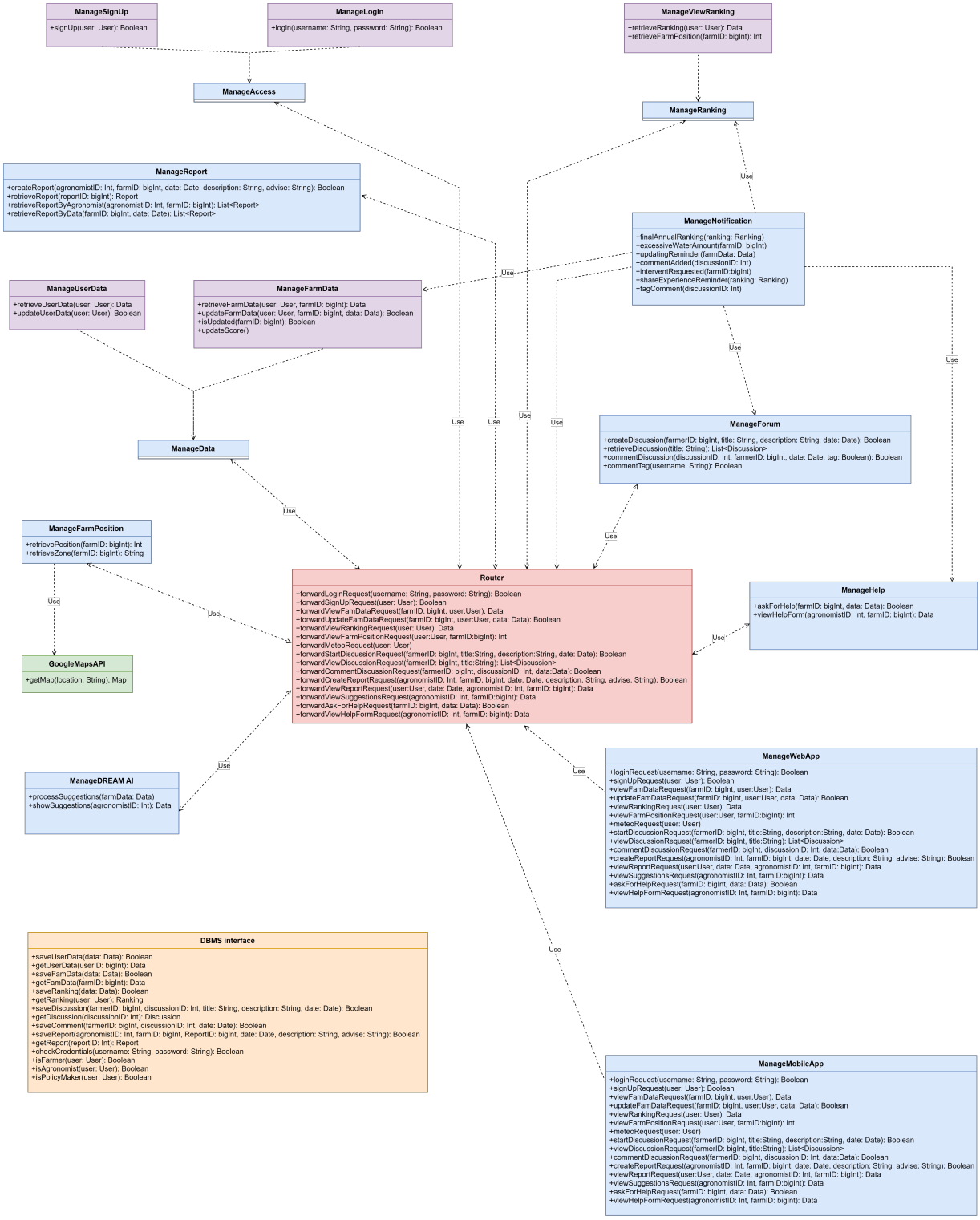


Figure 4: Component interfaces diagram using UML

The figure above, consistent with what is represented in the Component diagram (Section 2), shows the component interfaces of the application server and their interactions. Each arrow in the figure represents a dependence relationship of the element from which the arrow starts with respect to the element in which it arrives, when bidirectional implies a mutual use and so a mutual dependence between the interfaces. For each interface the main methods it allows to invoke are listed; however, it should be emphasized that these methods are not intended to coincide with the methods that developers will write, they are simply a logical representation of what component interfaces have to offer.

Some of the main characteristics of the interfaces represented are highlighted below:

- Let us first consider the DBMS, whose interface is shown in the figure disconnected from the others to reduce the number of arrows represented and therefore make everything clearer; in reality, as represented in the Component diagram, each component of the system interacts with the database by passing through the Data encryption / decryption, therefore, almost all the interfaces represented should be connected to the DBMS interface. As can be seen, it has some get methods that involve the extraction of information from the database and some save methods that, on the contrary, involve storing the data passed by parameter in the database. While the various get methods have the extrapolated data as a return value, the save methods have a boolean as a return value that will be marked true only if the information has been entered successfully. In addition to the methods described, there is a checkCredentials method used to allow registered users to log in to the system, it is a method that searches for the username passed by parameter in the database and if found checks if it matches with the password passed by parameter; if the login is successful then the boolean it returns is set to “true”, otherwise it will be set to “false”. Finally, there are three auxiliary methods that are isFarmer, isAgronomist and isPolicyMaker used to check, when necessary, the category to which a certain logged in user belongs.

- The ManageWebApp and ManageMobileApp interfaces are identical because each user must be able to use all the functions and services of the system regardless of the access method chosen. The methods that these interfaces have correspond to requests for services which, depending on the case, may involve the display of data stored in the database then they have to return data, or they can start processes such as the login request, the sign up or updating of the data just to name a few. In the second case it was decided to verify the correct execution of the requested process by setting a boolean as return values of the methods in question which will be "true" at the end of processing only if the process has been successful. Finally, it is important to underline that in these interfaces, as in many others, it is possible to distinguish methods linked to services that can be requested by any user registered and logged into the system, be it a farmer, an agronomist or a policy maker; for this reason, they are methods which have a generic user as parameter. On the other hand, there are methods that are linked to services reserved only for a specific category of users, such as the methods associated with the forum which are reserved for farmers; this second type of method requires an ID belonging to the specific category of users to which they are addressed to be passed as a parameter.

- The Router plays a central role in the system, all the component interfaces of the application server are connected to it as well as ManageWebApp and ManageMobileApp. Its main task is to forward requests from clients to the appropriate components of the server and then return the answers obtained at the end of the processing, this explains why its methods coincide in the signature with those of ManageWebApp and ManageMobileApp with the only addition of the term forward to indicate the action taken on such requests. ForwardMeteoRequest, namely the method to handle the request of viewing weather forecasts, is an exception because instead of being forwarded, this request is managed directly by the Router with the redirection of the user to the TSDPS site.

- One of the main component interfaces of the system is ManageData which is divided into two

sub-interfaces, one responsible for operations on user data and the other for operations on farm data, respectively ManageUserData and ManageFarmData. The latter has one method, retrieveFarmData, through it is displayed the data of the farm whose ID has being passed by parameter and updateFarmData through which the existing farm data are updated by adding new information passed as always by parameter to the method. ManageUserData has two methods similar to these ones, namely retrieveUserData to satisfy the request to view data from a logged in user and updateUserData to satisfy the update request instead. Finally, it was decided to equip ManageFarmData with two additional accessory methods: isUpdated is a method used to check every day how recent is the latest available data associated with the farm, if the data was not inserted during that day then the boolean return value is marked as "false", otherwise if they have been updated, the method will return "true"; updateScore is instead the method / routine that is invoked every time the data is updated to consequently update the farm score as well.

- ManageAccess is divided into two sub-interfaces as well, one responsible for login requests and the other for satisfying sign-up requests, respectively ManageLogin and ManageSignUp. Both of these interfaces have only one method which returns a boolean return value and this value will be marked as "true" only if the process in question (login or sign up) is successful.

- ManageRanking has a single sub-interface that is ManageViewRanking which deals with managing the requests for viewing the ranking that arrive. It is a simple interface with only two methods: retrieveRanking which precisely returns the requested ranking and retrieveFarmPosition which, given the ID of a certain farm passed by parameter, returns an integer corresponding to its position in the ranking.

- ManageHelp is the interface aimed at managing requests for help sent by farmers; therefore, it has the task of loading the requests with the relative content into the system database and then allowing them to be viewed by the agronomists called to intervene. The loading of the request is managed by askForHelp starting from the set of data to be inserted and from the ID of the requesting farm both passed by parameter, once again it is a method that returns a boolean value marked to "true" only if the process is successful. The display of the help request is instead handled by the other interface method, namely viewHelpForm.

- ManangeReport is the interface aimed at managing requests for viewing the reports and for their loading by agronomists. The insertion of a new report is managed by the CreateReport method and uses the set of data that is passed to it by parameter, then it returns a boolean value marked "true" only if the report has been loaded. To search for a report in the system, three possible ways have been imagined, corresponding to three possible methods: a first possibility is to search directly for the report with its ID, this is what the retrieveReport method does returning then the single searched report if it is found; alternatively, if you do not know the Report ID, you can try to find the desired report anyway with retrieveAllFarmReport or with retrieveReportByData. These two methods are one the generalization of the other, in fact, retrieveAllFarmReport is a method that returns the set of all reports made on the farm whose ID is passed by parameter, while retrieveReportByData in addition to the ID of a farm also receives a date as parameter and uses it to filter the list of reports displayed.

- ManageDREAM AI is the interface that manages all the operations required of DREAM AI. It has only two methods: the first is processSuggestions which, as the name indicates, takes care of processing the farm data received by parameter, obtaining appropriate advice to offer; the second method is showSuggestions which takes care of showing the processed data to the authorized agronomist whose ID is passed by parameter.

- ManageForum is the interface aimed at managing all the operations on the forum; in particular, therefore, it must provide for the creation of new discussions, the search for already existing ones,

the creation of comments and manage possibly inserted tags. The loading of a new discussion into the system is managed by createDiscussion and takes advantage of all the information passed to it by parameter, then returns a boolean value marked “true” only if the discussion has been loaded. The search for a pre-existing discussion in the system is managed by the retrieveDiscussion method and returns the set of all the discussions having as title the string passed by parameter to the method. The possibility of creating a comment on a discussion is managed, on the other hand, by the commentDiscussion method which receives as a parameter the ID of the discussion in question, the data to add and a boolean value that indicates the presence or not of a tag directed to another user in the comment itself; the method has a boolean return value which will be marked “true” only if the comment has actually been added to the discussion. If the boolean tag parameter of commentDiscussion is set to “true”, then the comment includes a tag to another user whose management will be delegated to another interface method, namely commentTag; this is a method that simply creates the desired tag on the user whose username is passed by parameter.

- ManageNotification is the interface aimed at managing the set of notifications circulating in the system. It is an interface consisting of methods / routines each of which is associated with a separate notification and is automatically invoked when the situation, identified through the timely communication between NotificationManager and one of the components to which it is connected, requires it.

- ManagePosition is a very simple interface of only two methods, aimed exclusively at giving information about the position of a certain farm. The first method it has is retrievePosition, which given a certain farm whose ID is passed by parameter returns an integer corresponding to its position. Moreover, it is also equipped with the retrieveZone method which, given a farm, returns a string indicating its belonging zone. In order to show the location of the requested farm more effectively, ManagePosition is the only server component interface that has the ability to interact with the external Google Maps API, GoogleMapsAPI and use the map processed by its getMap method.

2.6 Selected architectural styles and patterns

As partly described in the overview, a three-tier client-server architecture has been used to develop the system. This choice promotes a major decoupling of the system, increasing the reusability, scalability and flexibility. Furthermore, components in the application server have been thought to be cohesive and with low coupling among modules to make the system more comprehensible and modifiable.

The communication protocol used to exchange messages is HTTPS. This is also endorsed by the importance that data plays in the system. In this way it is possible to reduce the coupling among client and server components. The Hypertext Transfer Protocol Secure (HTTPS) takes advantage of the Secure Sockets Layer (SSL) to create an encrypted stream of data using Diffie – Hellman key exchange. In this way the security of the data transmitted between client and server is guaranteed by the impossibility of decrypting the data.

Furthermore the system uses the cache on the client side: this allows to avoid some interactions between the client and the server, speeding up the communication. However, since lots of data needs a real-time update (ranking, farm data, reports), the *If-Modified-Since*[9] header will be used (in client request) to check the consistency of the file in the cache.

Regarding the format in which the data are transmitted, JSON is used of its simplicity. It is less verbose and this makes it more readable and allows a much faster parsing. The JSON file is transmitted over the network via XmlhttpRequest.

Regarding authentication a token-based approach is adopted. The client-side user state is then saved via JSON Web Token (JWT), which defines a way of securely transmitting information between a client and a server. Every 2 hours the user is asked to re-log in, for security reasons.

With respect to the database server, instead, the application server acts as a client making queries and waiting for response. Data is validated to predict SQL injection before requests.

2.6.1 Three Tier Client-Server

A multilayered architecture is a client--server architecture in which presentation, application and data access functions are physically separated. We choose a multitier architecture because it allows to decouple the complexity of the system, making it more flexible and reusable. Indeed, the developers acquire the power of modifying or adding specific layers without disrupting the entire application. Moreover, this kind of architecture allows to separate completely the access to data from the layer where the logic for presentation and for the interaction with the customers is located: this is very important to make the system safer since the treated information are sensitive data. More precisely, we adopted a three tier architecture, composed of a Presentation (P) tier, an application (A) tier and a data access (D) tier: the mentioned separation between the clients and the data is possible thanks to the A tier.

To perform the application the following pattern are used:

2.6.2 Façade

This pattern is used from the "Router" component to provide an interface to the client using which the client can access the system. A facade is an object that serves as a front-facing interface masking more complex underlying or structural code. In addition, in this pattern the client interacts with a simpler interface instead of an intricate one thanks to the hiding of complexity of the larger system.

2.6.3 Mediator

The "Router" component, in addition to providing a unified interface to the user, also acts as a "mediator" within the server, in order to group all the main functions of each component and make the code more modular.

An advantage of this choice is that in the event of a change to a component's functionality, only the interface that this component has with the router needs to be changed. Furthermore, having only one connection is also easier to understand the consequences that a certain change creates within the server, thus avoiding hidden and unclear connections.

The only component that bypasses the router is the NotificationManager.

2.6.4 Observer

For each notification, this type of pattern was chosen to avoid wasting resources (clock cycles) in the server. Each user is logged in as "observers" and at the time of a communication, the "NotificationManager" will perform the function of "subject entity" which will notify the group of users concerned, in some cases all users (notification of final ranking), in other cases only some (reminder notification to upload farm data to farmers only).

2.6.5 Singleton

This pattern was chosen for ranking, as all components that use or display it must have the same instance of this class.

2.6.6 MVC

Carrying on the description of the client we follow the Model-View-Controller (MVC); This architectural pattern impose to separate the application in 3 different components permitting a full encapsulation with the consequence that each component can be modified in a independent way with the respect of other ones; furthermore this permits an easier way to perform testing operations.

2.7 Other design decisions

2.7.1 Thin Client

In order to have a client as light as possible the application follows the ”Thin Client” paradigm; in this way the client doesn't need to perform big computations but instead it must handle only minimal efforts in order to communicate with the application servers.

Moreover, this paradigm permits the benefit of an easier synchronization of data across multiple entities and an easier way to update the application (the client doesn't need to change so much even if the change, with the respect to the previous behavior, is huge).

2.7.2 Firewalls

Talking about security instead we make use of firewalls in our architecture. In particular we have an actual firewall protecting the access to our DBMS, for the application servers security we leave at the reverse proxy the task. These components permit to protect the internal network of the System from possible attacks by the untrusted external network; furthermore they operate by letting a possibility for filtering packets that pass through them.

2.7.3 Relational databases

Relational databases are a good choice when there is the need to deal with several transactions and when the data are linked by some relationships (users, third parties, runs etc.). This fits for our purposes, which can be cataloged as data analysis, and, even if for some reasons (for example a huge collection of data) a non--relational database would be more performing, a relational one has been chosen because it is necessary to create complex and dependent data structures and use a very expressive and known query language as SQL.

2.8 DREAM AI Algorithms

2.8.1 Explanation

The main goal of DREAM AI is to help agronomists in giving advice to farms. To do this, the system plans to use the information stored in the database to find similarities with products grown by similar farms that have given good results.

DREAM AI is a component that includes several parts, first of all the one that extracts the information of the farms located in the same province as the one to be recommended from the database, which the recommender model will use. This part will not be described in detail and has the goal of generating the *data_ FSM.csv* file.

The Farm Score Matrix (FSM) is a matrix that contains in the rows all the farms present in the province of the farm to be recommended, while in the columns all the products in the system. In each cell there is the latest $\frac{c_i}{s_i} * F_i$ of a farm that has collected a product (where i is the type product (column), c_i is the quantity collected product ($gr.$), s_i is the size of the land where the product was grown (m^2) and F_i is a coefficient related to the type of food). If the farm has not grown product i for more than a year, or has never grown it, the relative cell will have the value 0.

We will use one of the most used and effective recommender systems is Collaborative Filtering.[7] In the "item based" version, in which items corresponds to products, the first step is to create a similarity matrix between the items (with the cosine similarity method), then we can estimate the score of an item (product) by adding the product between the score and the similarity of the k items most similar to that item:

$$\tilde{score}_{fp} = \frac{\sum_{j \in KNN(p)} score_{fj} \cdot s_{jp}}{\sum_{j \in KNN(p)} s_{jp}}$$

where $\tilde{score}_{fp}$ is the estimation of farm f and product p, $KNN(p)$ are the k-nearest neighbors items of p (k similar), $score_{fj}$ is the score of farm f for product j, s_{jp} is the similarity between product j and p.

So based on the number of farms that have performed well from two products, their level of similarity is deduced. After that, using the information of the product scores already used by the farm, it is possible to estimate whether the score of a certain product will be good or not.

2.8.2 Code

Classes already written by Maurizio Ferrari Dacrema[8] have been used.

```
################################## IMPORTS ##################################
import pandas as pd
import numpy as np

from ItemKNNCFRecommender import ItemKNNCFRecommender

################################## READ DATA ##################################

FSM = pd.read_csv(filepath_or_buffer="data_ FSM.csv",
                          sep=',',
                          names = ["farm_id", "product_id", "score"],
                          header=0,
                          dtype={'row': np.int32, 'col': np.int32, 'data': np.float64})

####################### ISTANTIATE AND FIT THE CF #######################

recommender = ItemKNNCFRecommender(FSM)
recommender.fit()

################################ PRODUCE CSV ################################

f = open("submission.csv", "w+")
f.write("farm_id,product_id\n")
farm_id = 729445
recommended_items = recommender.recommend(farm_id, cutoff=5, remove_seen_flag=True)
well_formatted = " ".join([str(x) for x in recommended_items])
f.write(f"{farm_id}, {well_formatted}\n")
```

Figure 5: File run.py

This are the two most important file, the other one are listed in the code folder.


```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on 23/10/17

@author: Maurizio Ferrari Dacrema
"""

from Recommender_utils import check_matrix
from BaseSimilarityMatrixRecommender import BaseItemSimilarityMatrixRecommender

from IR_feature_weighting import okapi_BM_25, TF_IDF
import numpy as np

from Compute_Similarity import Compute_Similarity


class ItemKNNCFRecommender(BaseItemSimilarityMatrixRecommender):
    """ ItemKNN recommender"""

    RECOMMENDER_NAME = "ItemKNNCFRecommender"

    FEATURE_WEIGHTING_VALUES = ["BM25", "TF-IDF", "none"]



    def __init__(self, URM_train, verbose = True):
        super(ItemKNNCFRecommender, self).__init__(URM_train, verbose = verbose)


    def fit(self, topK=50, shrink=100, similarity='cosine', normalize=True, feature_weighting = "none",
URM_bias = False, **similarity_args):

        self.topK = topK
        self.shrink = shrink

        if feature_weighting not in self.FEATURE_WEIGHTING_VALUES:
            raise ValueError("Value for 'feature_weighting' not recognized. Acceptable values are {},
provided was '{}'".format(self.FEATURE_WEIGHTING_VALUES, feature_weighting))

        if URM_bias is not None:
            self.URM_train.data += URM_bias

        if feature_weighting == "BM25":
            self.URM_train = self.URM_train.astype(np.float32)
            self.URM_train = okapi_BM_25(self.URM_train.T).T
            self.URM_train = check_matrix(self.URM_train, 'csr')

        elif feature_weighting == "TF-IDF":
            self.URM_train = self.URM_train.astype(np.float32)
            self.URM_train = TF_IDF(self.URM_train.T).T
            self.URM_train = check_matrix(self.URM_train, 'csr')

        similarity = Compute_Similarity(self.URM_train, shrink=shrink, topK=topK, normalize=normalize,
similarity = similarity, **similarity_args)


        self.W_sparse = similarity.compute_similarity()
        self.W_sparse = check_matrix(self.W_sparse, format='csr')
```

Figure 6: File ItemKNNCFRecommender.py

3 User interface design

3.1 Prototypes

This section is dedicated to the graphic prototypes of the user interface. Some are shown both from the app and the website. We remind you that the interfaces are similar to facilitate their use and that the two access points allow each user to perform the same functions.

Data of a farm are shown to its owner in Fig.11. As already explained, every time a farmer wants to enter the daily data he must first visit this screen and then press the edit button (down to the right) to enter the "edit mode".

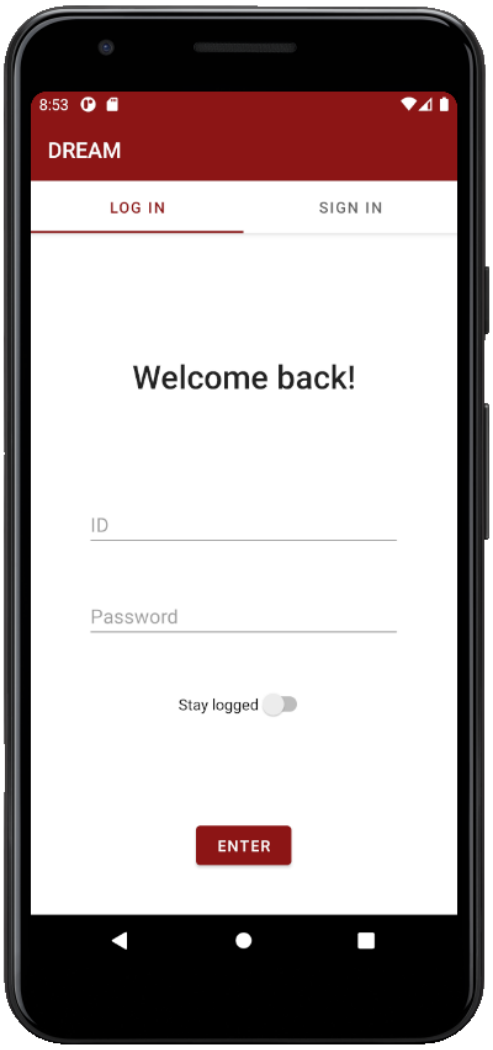


Figure 7: App - Login screen

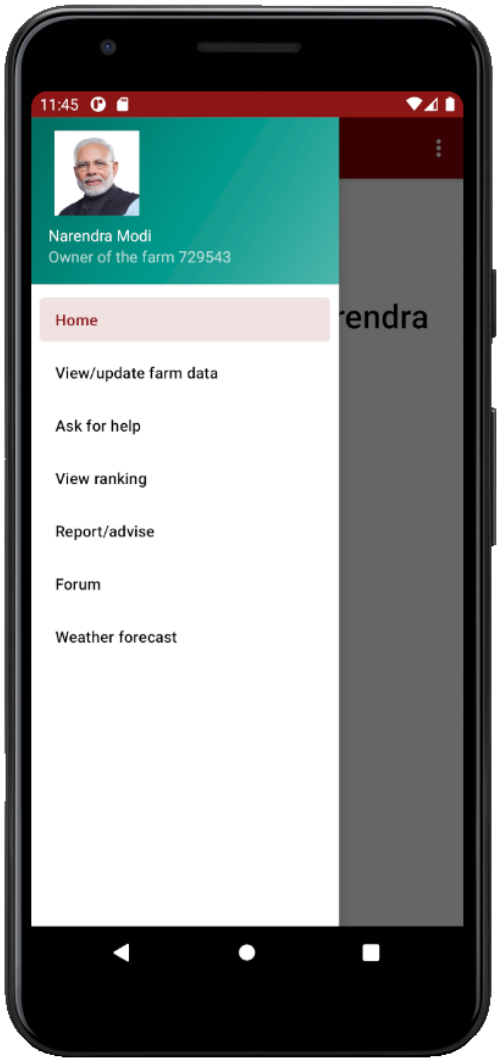


Figure 8: App - Menu

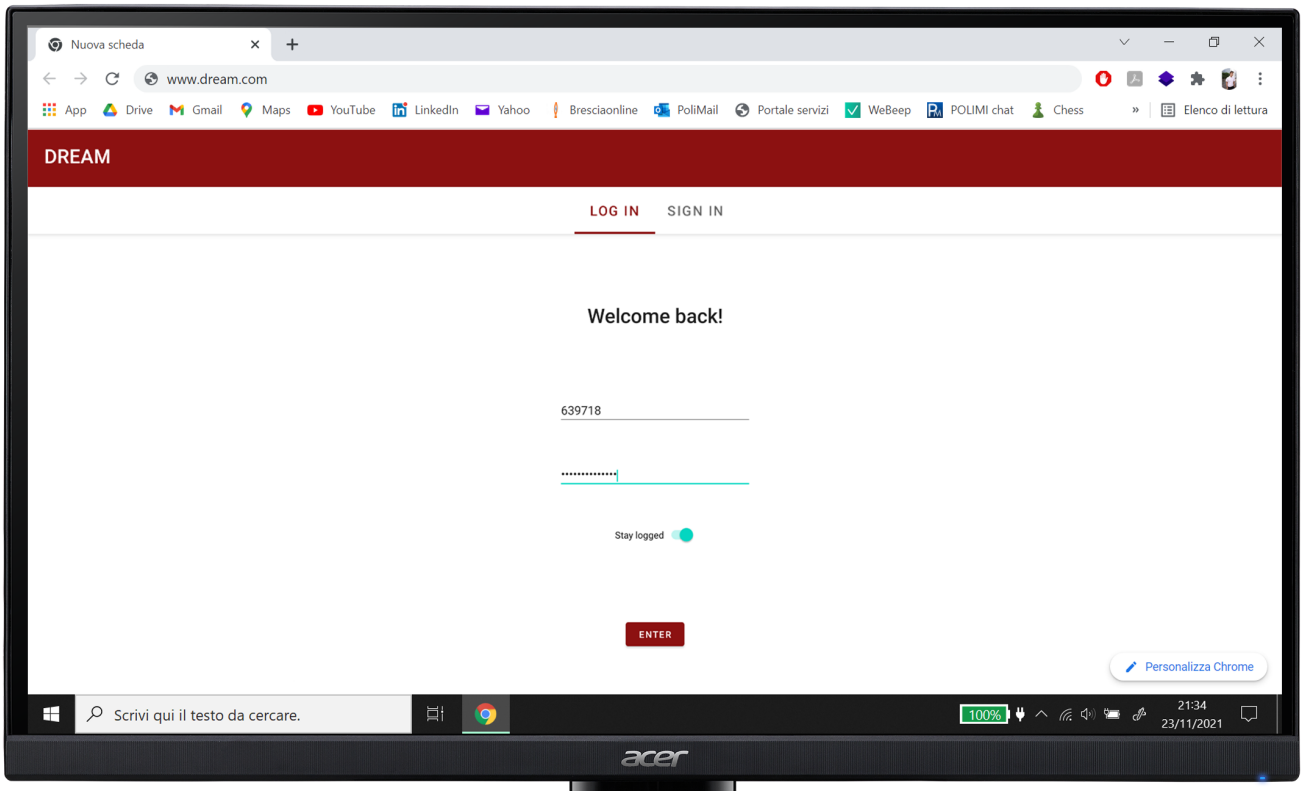


Figure 9: PC - Login screen

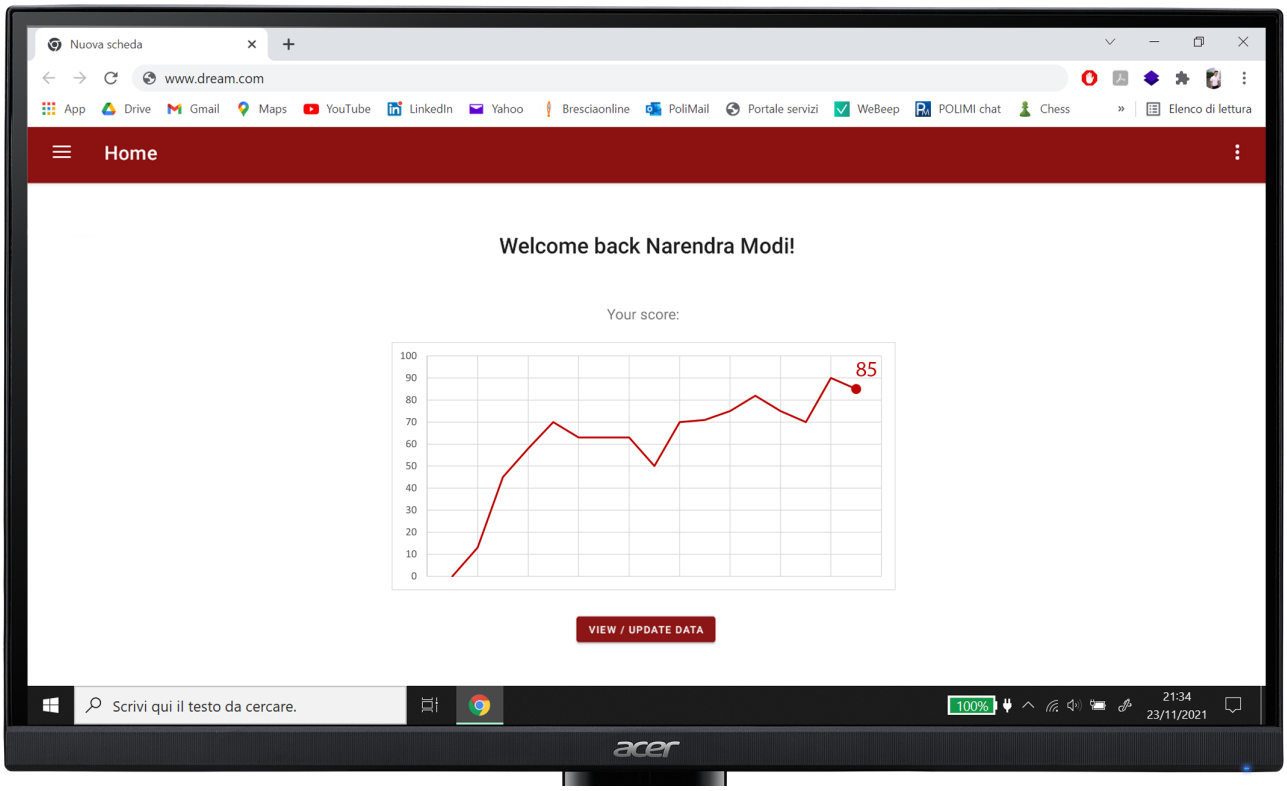


Figure 10: PC - Home screen

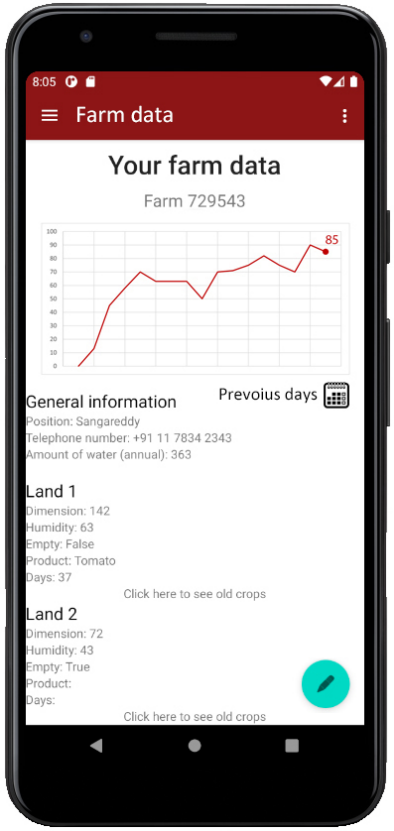


Figure 11: App - Farm Data

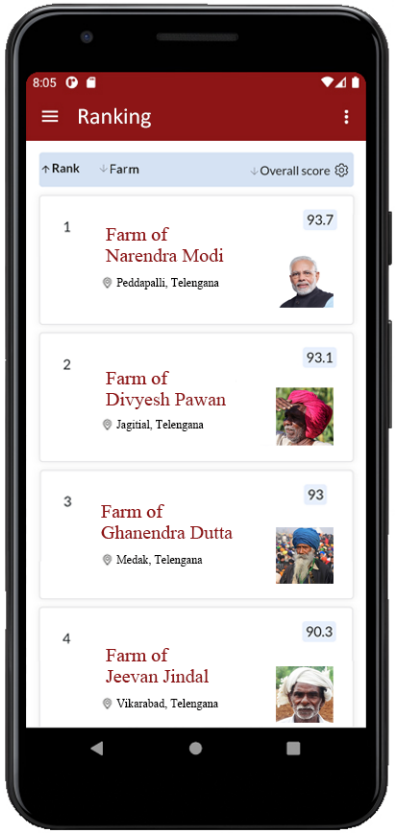


Figure 12: App - Ranking screen

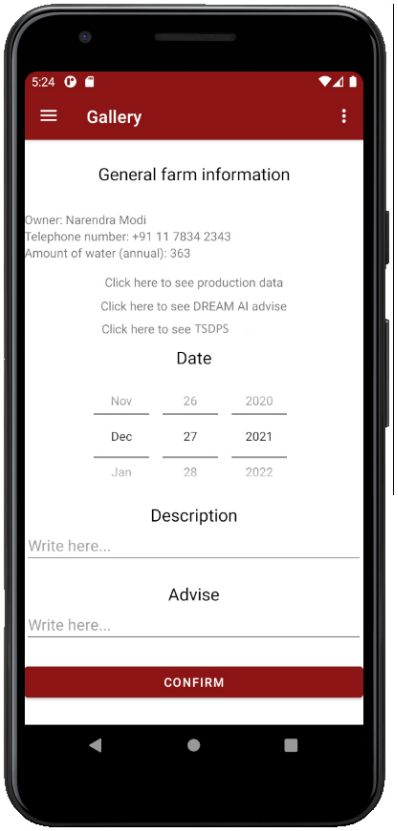


Figure 13: App - Report screen

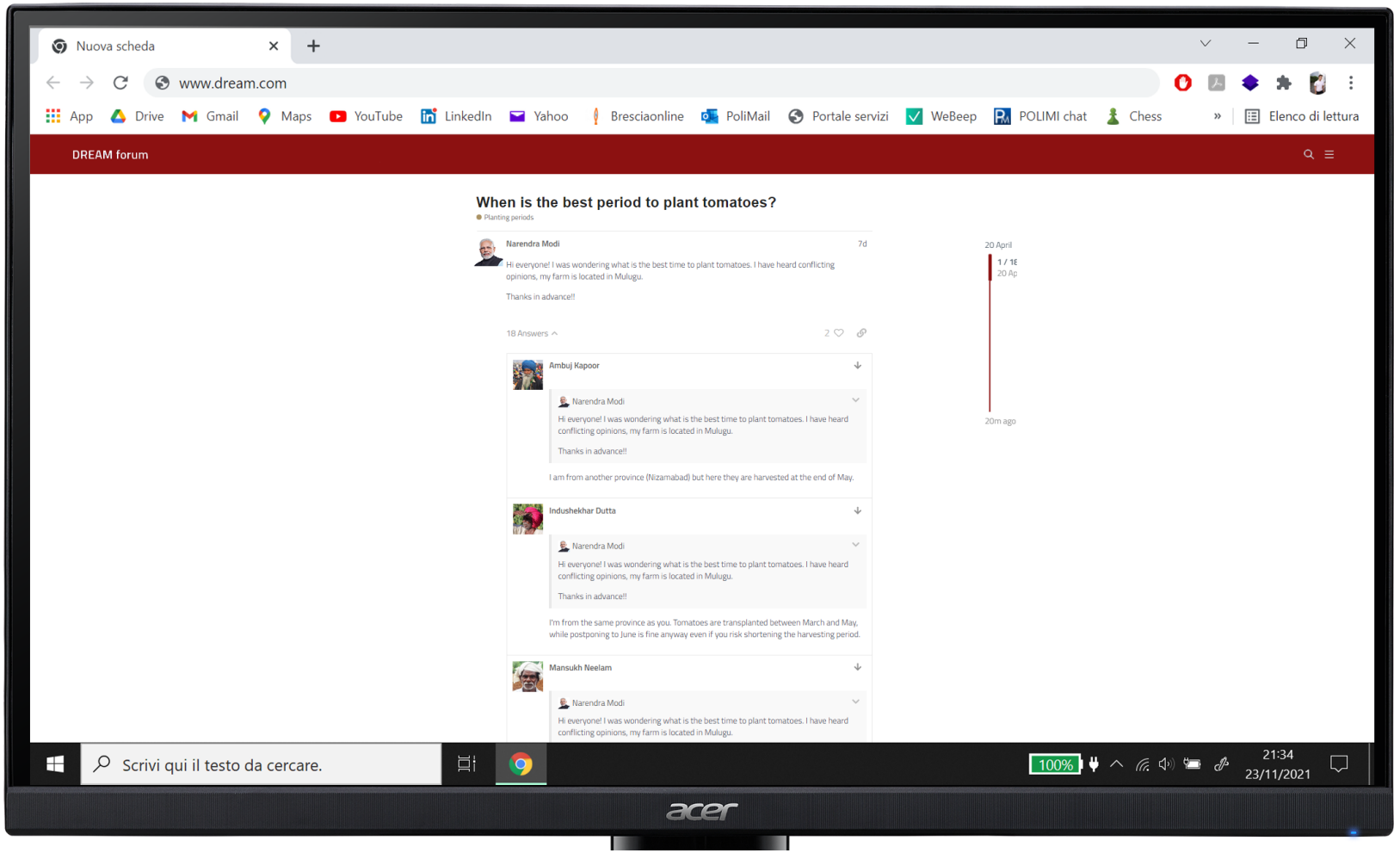


Figure 14: Webapp - Forum screen

4 Requirements trace-ability

This section bids the goals specified in RASD with design components. Each component is listed only once, even if it is used more than once.

G1	Allow policy makers to identify the farmers to be rewarded and those performing bad
	Requirements: **R1,R2,R4,R5,R11,R21,R22**
	Components: • UserMobileApp/WebApp • Router • AccessManager [LoginService] • RankingManager [ViewRankingService] • DataManager [FarmDataService] • DataManager [UserDataService] • Data encryption/decryption • DBMSSService
G2	Allow farmers to ask the help they need
	Requirements: **R6,R7,R8**
	Components: • UserMobileApp/WebApp • Router • AccessManager [LoginService] • DataManager [FarmDataService] • RankingManager [ViewRankingService] • HelpManager • NotificationManager • Data encryption/decryption • DBMSSService

G3	Allow policy makers to evaluate if the advise given by agronomists works Requirements: **R4,R5,R9,R10,R11,R12,R13,R21,R22** Components: • UserMobileApp/WebApp • Router • AccessManager [LoginService] • GoogleMaps • DataManager [FarmDataService] • RankingManager [ViewRankingService] • ReportManager • Data encryption/decryption • DBMSSService
G4	Being able to anticipate climatic phenomena Requirements: **R14** Components: • UserMobileApp/WebApp • Router • AccessManager [LoginService] • Router [Redirect] • TSDPS
G5	Allow farmers to visualize relevant data based on their location and type of production Requirements: **R11,R17,R18,R19** Components: • UserMobileApp/WebApp • Router • AccessManager [LoginService] • DataManager [FarmDataService] • Data encryption/decryption • DBMSSService

G6	Allow farmers to keep track of their production data Requirements: **R16,R20,R21,R22,R23,R31** Components: • UserMobileApp/WebApp • Router • AccessManager [LoginService] • DataManager [FarmDataService] • NotificationManager • Data encryption/decryption • DBMSSService
G7	Allow farmers to create discussion forums with the other farmers Requirements: **R24,R25,R26,R27,R28,R29,R30** Components: • UserMobileApp/WebApp • Router • AccessManager [LoginService] • ForumManager • NotificationManager • Data encryption/decryption • DBMSSService
G8	Encourage the best farmers to share their experience on the forum Requirements: **R3,R21,R22** Components: • NotificationManager • RankingManager [ViewRankingService] • UserMobileApp/WebApp • Router • AccessManager [LoginService] • ForumManager • Data encryption/decryption • DBMSSService

G9	Allows agronomists to send production advice to farmers and write reports on their visits Requirements: **R8,R21,R22,R32,R33** Components: • NotificationManager • UserMobileApp/WebApp • Router • AccessManager [LoginService] • DataManager [FarmDataService] • RankingManager [ViewRankingService] • ReportManager • DREAM AI • Data encryption/decryption • DBMSSService
G10	The system must distinguish users in 3 categories: farmers, policy makers and agronomists Requirements: **R35,R36,R37** Components: • UserMobileApp/WebApp • Router • AccessManager [LoginService] • DataManager [UserDataService] • Data encryption/decryption • DBMSSService
G11	The system must give advice to agronomists (DREAM AI) Requirements: **R15,R23,R34** Components: • DREAM AI • ReportManager • DataManager [FarmDataService] • Data encryption/decryption • DBMSSService

R1	All users are notified when the annual ranking is completed
R2	The system must allow policy makers to view the data needed to contact farmers
R3	The best farmers have to be notified asking them to share their story in the forum
R4	Policy makers must be able to access sensitive data of other users
R5	Policy makers must be able to view farm production data
R6	Agronomists must be notified if their help has been required
R7	The system must allow agronomists to view the data needed to contact farmers
R8	Agronomists must be able to access to sensitive data of other users
R9	Politicians can view the graph of each farm's score trend
R10	The system must show the reports written by agronomists with the advice given
R11	The system must show the ranking of the farms
R12	The system must show the map of the farms contained in each zone
R13	The system must show the agronomist associated to each zone
R14	The system must allow users to be redirected to TSDPS meteo
R15	DREAM AI must process suggestions based on the evolution overtime of a farm's production and on other farms results
R16	The system must store the data entered by the farmer in chronological order from the most recent to the least
R17	The system must display the data entered by the farmer in chronological order from the most recent to the least
R18	Farmers must be able to view their farms' production data
R19	Farmers must be able to view their score
R20	Farmers can update their farms' production data
R21	The system must update the score associated to a farm every time its data are updated
R22	The system must update the ranking of the farms
R23	DREAM AI has to reprocess the projections and suggestions associated to each farm when its data are updated
R24	The system must allow farmers to create a new discussion on the forum
R25	A farmer must be able to comment on a forum discussion

R26	A comment from a user must be able to tag another user
R27	The system must send a notification to a user when when his discussion is commented on
R28	The system must send a notification to a user when tagged in a comment
R29	The forum must store the discussions/comments in chronological order
R30	The forum must display the discussions/comments in chronological order
R31	The system must send a notification to the farmers (at 21.00) to remind them to update the daily data
R32	Agronomists must be able to view farm production data
R33	Agronomists must be able to write reports in the system
R34	DREAM AI must propose the advices computed to agronomists
R35	The system must show a specific different form to compile to each user, based on the category he selects, during registration
R36	The system must ask each user to fill in the form to complete the registration
R37	Each user can view only the features specific to his category

5 Implementation, integration and test plan

5.1 Overview

This last part is very important because it is almost impossible to develop error free software, so the phase of verification and validation takes an important role. Program testing can be used to show the presence of bugs, but never to show their absence. Thus, for this reason, the aim is to find as many bugs as possible until the release date of the application.

5.2 Implementation Plan

The system and its constituent subsystems must be implemented, tested and integrated using a bottom-up approach. We therefore opted for a gradual approach to be followed in the implementation of the system which first involves the implementation of the lower components up to the top, in this way the system is implemented incrementally and is test driven because it allows to conduct tests on the components in parallel with their implementation. The chosen approach is followed for both the server and client parts of the system and the two will be implemented and tested in parallel. The main factors considered in drawing up the plan for the implementation of the system components are:

1. the importance that each component plays in the user experience
2. the dependencies it has with respect to other components.

To analyze the importance of individual components and corresponding services in the system, it was decided to first consider the main features offered by DREAM to its users.

Feature	**Importance for the customer**	**Difficulty of implementation**
Sign up and login	High	Low
Visualize/update user data	Medium	Low
Visualize/update farm data	High	Medium
Create/view report	Medium	Medium
View/update ranking	High	Medium
Create/view discussion and comment	Medium	Medium
View meteo	Medium	Low
View map and position	Low	High
Ask for help	High	Medium
Process and propose suggestions (DREAM AI)	Medium	High
Manage notifications	Low	Low

The table above highlights for each system functionality the value it has for the user and estimates the difficulty requested by its implementation between three possible alternatives Low, Medium and High. From the importance attributed to these features and knowing which system components and services provide them, we can deduce the importance to be attributed to each component of the application/web server of the system during implementation. Specifically, we can note that the management of notifications delegated to the NotificationManager is not of great importance for the user because it is secondary to other functions offered by DREAM. Similarly, the location and map display also plays a marginal role compared to other services, which is why they are of little importance to the user although difficult to implement because they require the system to interface with the external Google Maps API. Most of the other functions such as the viewing and updating of user data entrusted to the UserDataService, the creation and viewing of reports entrusted to the ReportManager, all the operations that have to do with the forum entrusted to the ForumManager and the visualization of meteorological data, they are all features of medium importance to the user. The work carried out by DREAM AI is also of medium importance, but it is complex to implement. Finally, we can note some features that instead stand out for having high importance for the user including sign up and login managed by the AccessManager, display and update of the farm data managed by DataManager, display and update of the ranking managed by RankingManager and management of farmer help requests entrusted to HelpManager. What has just been described translates into the higher priority given to the implementation of AccessManager, DataManager, RankingManager, HelpManager and any internal services that compose them.

As regards the second factor considered in drawing up the plan, namely the dependencies between different components of the system, we can first of all note that almost all the components of the system interact with the DBMS because they require access and query on the database in order to store or retrieve information from it. In addition, it should be noted that access to the DBMS always requires the intervention of the Data encryption / decryption component, therefore, given their great utility, these will be the first two elements to be implemented. On the other hand, the Router, as already mentioned, has the fundamental role of forwarding client requests to the server and then returning the responses but also of coordinating the activity of the entire system by putting the other server components in communication with each other. However, the router does not possess any kind of business logic, for this reason it will have to be implemented as the last element. We observe that the NotificationManager depends in its functioning on the DataManager, the ForumManager, the HelpManager and the RankingManager, therefore, it will have to be implemented after them. Similarly, the ForumManager, the RankingManager and the ReportManager all depend on the DataManager and the AccessManager, while the HelpManager in addition to the previous ones also depends on DREAM AI. Finally, it is important to note that the DataManager depends on the AccessManager as well.

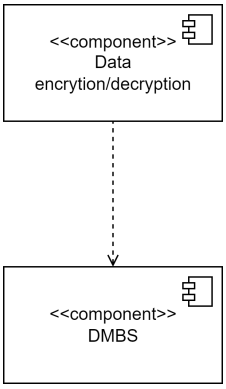
The set of dependencies described above and the priorities previously assigned to each component of the application / web server of the system according to the importance for the user of the functions it guarantees, allow us to establish an order to follow in the implementation of these components:

1. DBMS
2. Data encryption/decryption
3. AccessManager
4. DataManager
5. RankingManager
6. DREAM AI
7. HelpManager
8. ReportManager

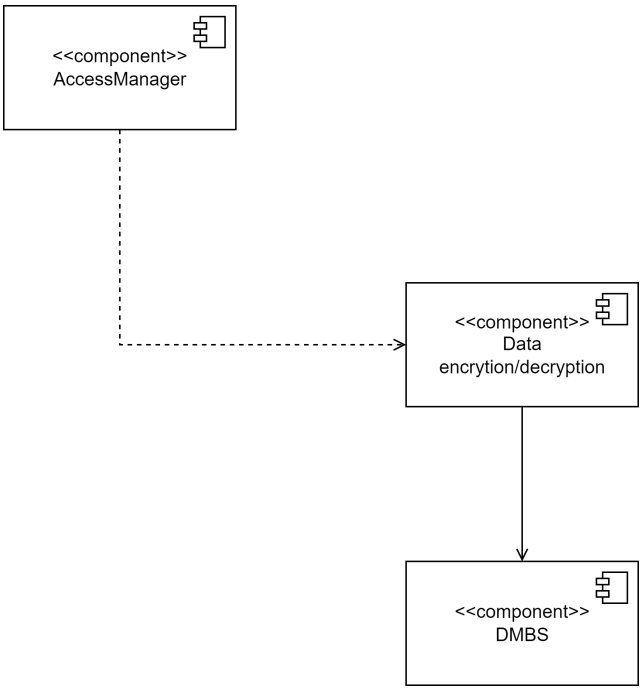
9. ForumManager
10. NotificationManager
11. Router

5.3 Integration Strategy

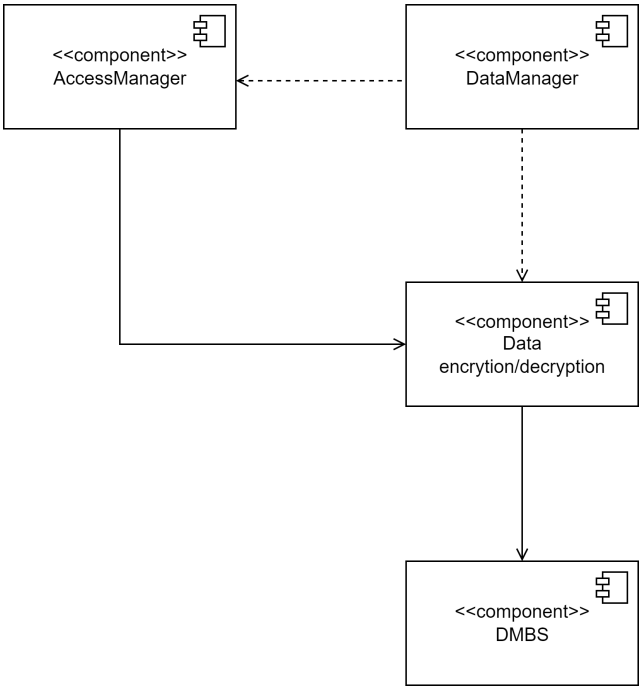
The following diagrams show step by step how each component of the system is implemented and integrated with the others, according to the order described above. In each diagram the arrows indicate dependency relationships between distinct components, such that the implementation of a certain component from which an arrow in the diagram starts takes advantage of the implementation of the component to which the arrow arrives. In addition, dashed arrows are used to describe the dependencies with respect to other system components of the element that is going to be implemented and integrated, while solid arrows indicate dependencies between already implemented and integrated elements.



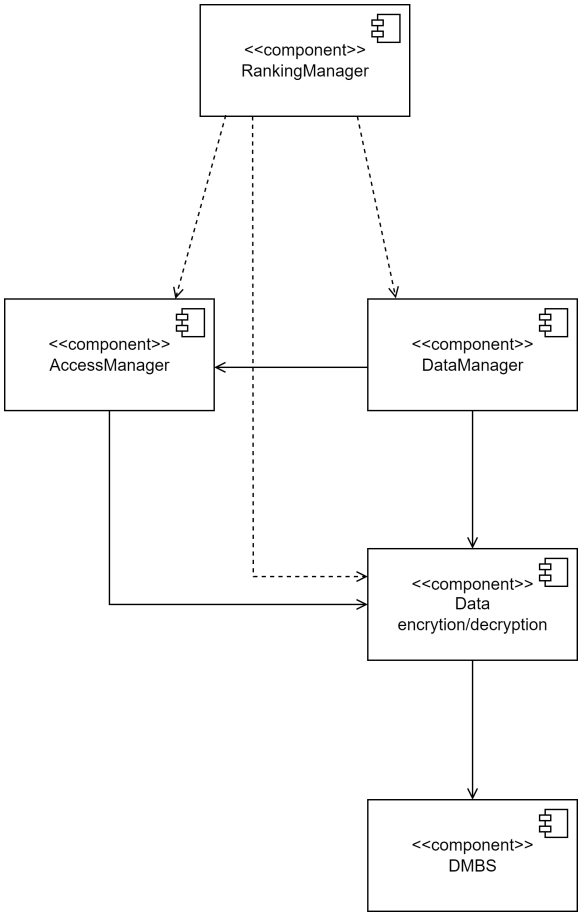
As already mentioned after being implemented and unit tested, the first elements that are integrated are the DBMS and the Data encryption / decryption.



The AccessManager is implemented and unit tested and then integrated with the first two elements.

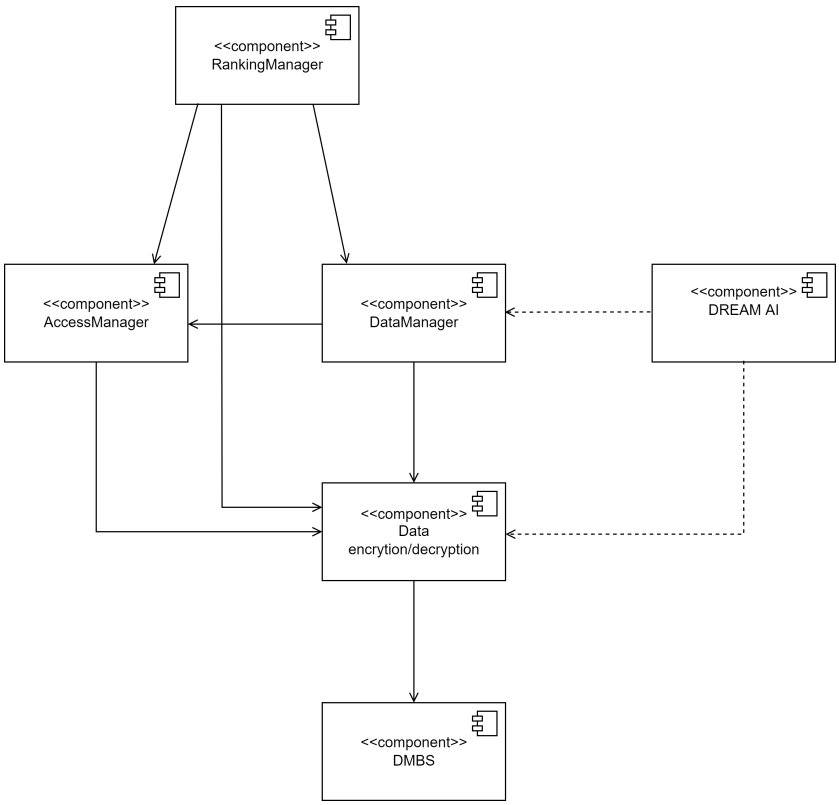


The DataManager is implemented and unit tested and then it is integrated with the already implemented system components.

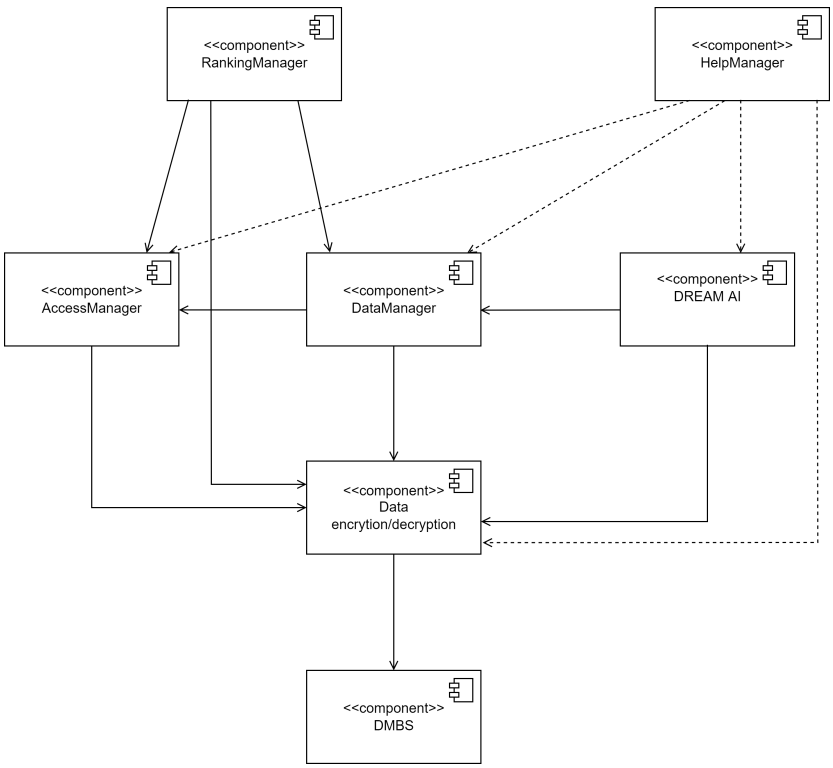


The RankingManager is implemented and unit tested and then integrated with the already implemented

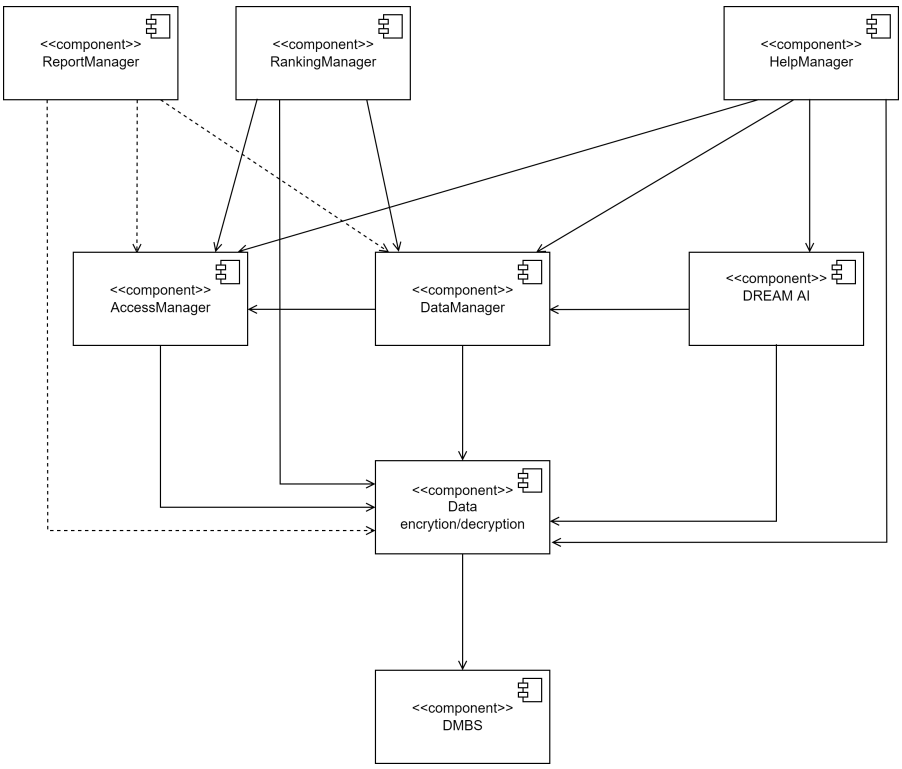
components of the system.



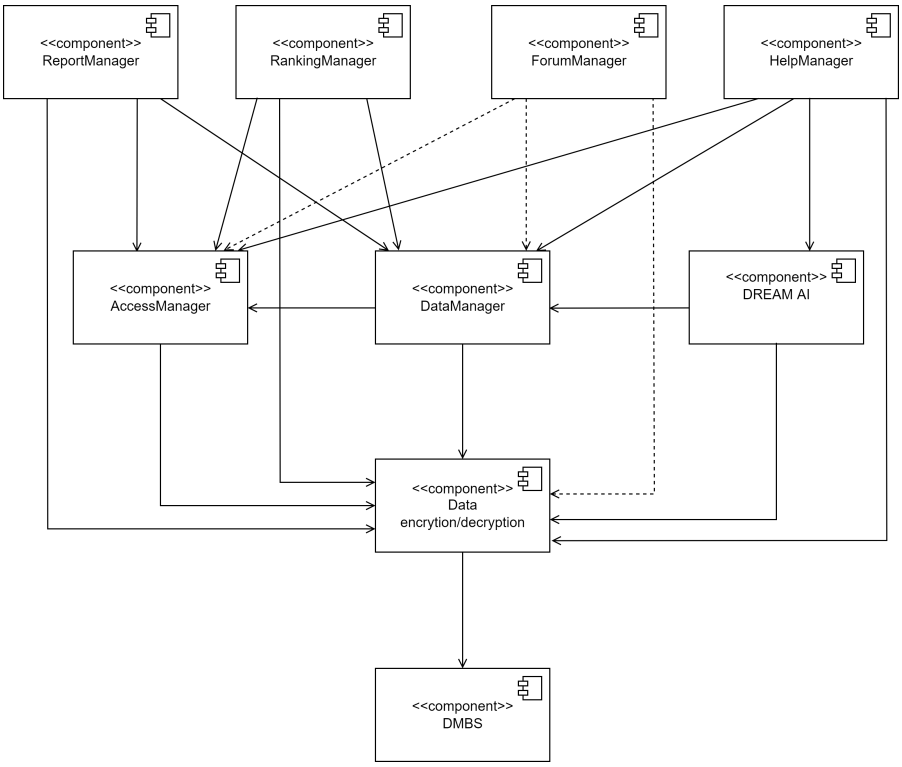
DREAM AI is implemented and unit tested and then integrated with the already implemented components of the system.



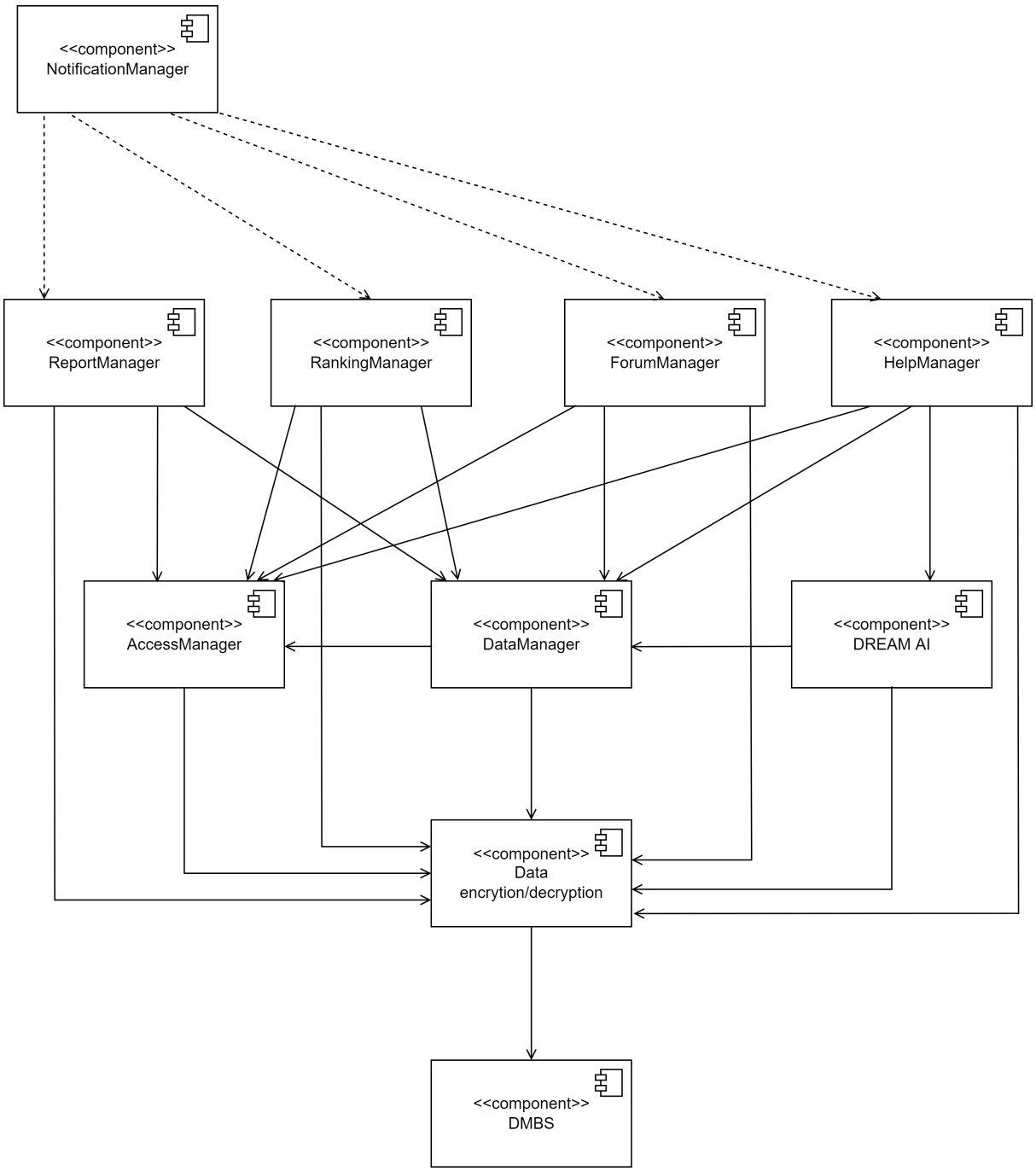
The HelpManager is implemented and unit tested and then it is integrated with the already implemented system components.



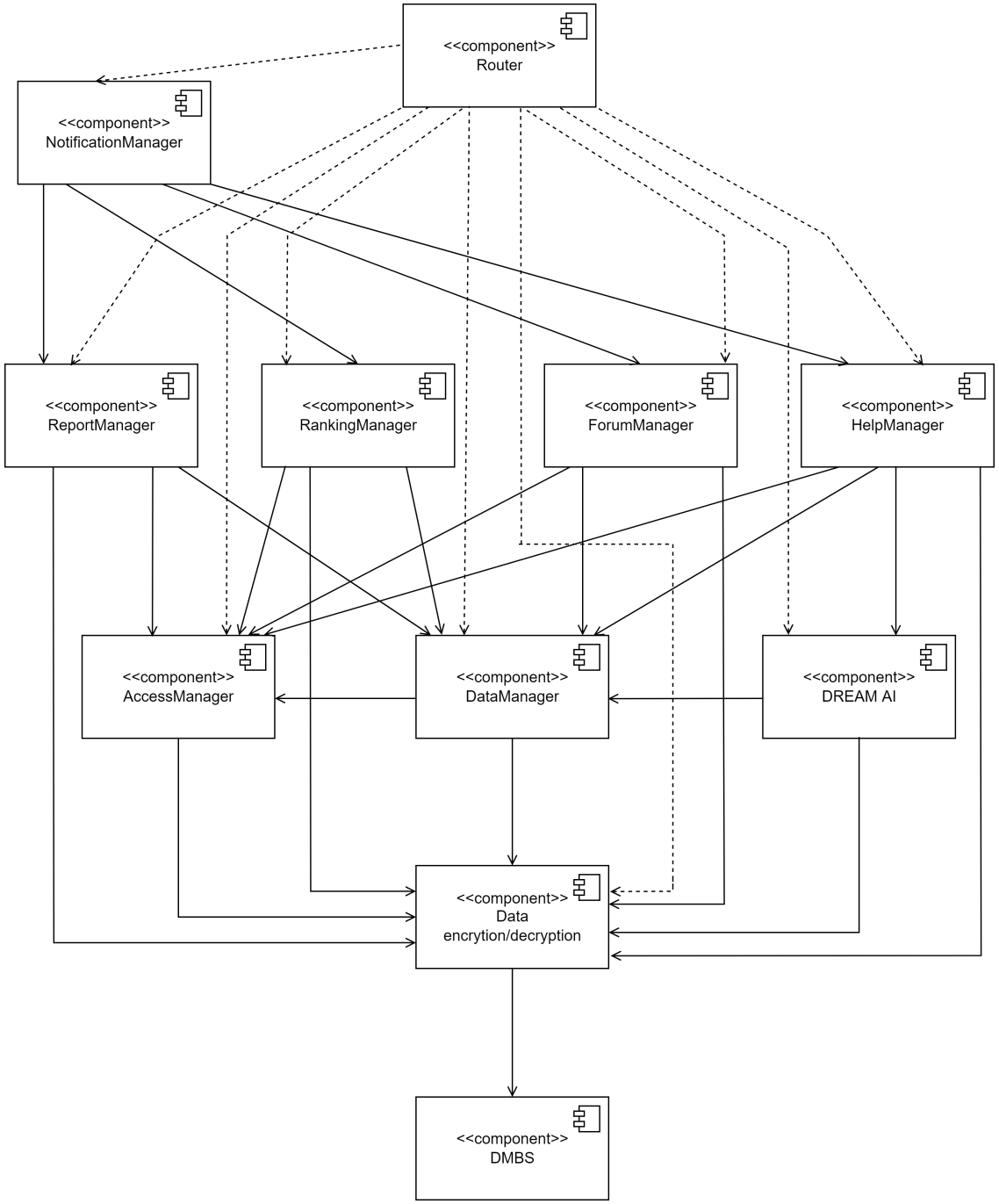
The ReportManager is implemented and unit tested and then it is integrated with the already implemented system components.



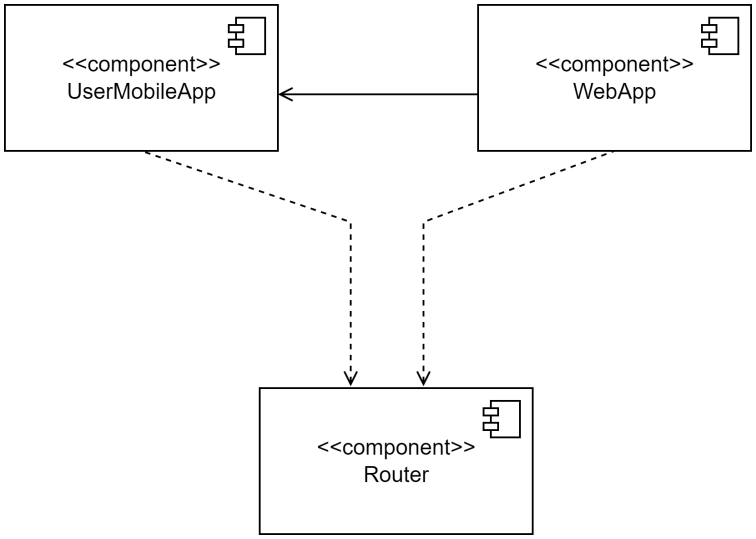
The ForumManager is implemented and unit tested and then it is integrated with the already implemented components of the system.



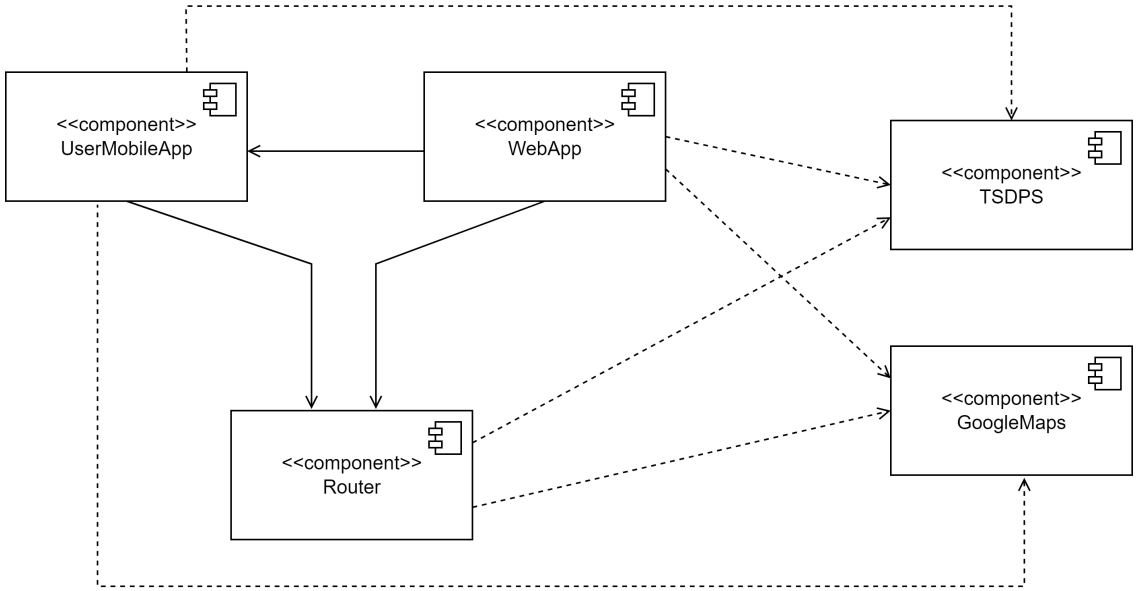
The NotificationManager is implemented and unit tested and then it is integrated with the already implemented components of the system.



The Router is implemented and unit tested as the last element of the server, then it is integrated with all the components already implemented in the system.



The implementation and testing of the client part of the system, ie UserMobileApp and WebApp, as already mentioned, is carried out in parallel with the development of the server. Once implemented and unit tested, UserMobileApp and WebApp are finally integrated into the system; specifically, integration is only possible when the implementation and testing of both the client and server sides has been completed and involves connecting the clients to the Router.



The last step in the implementation and integration of the system involves its connection to the external services of TSDPS and Google Maps; specifically, the WebApp, the UserMobileApp and the Router will be connected to the external components.

5.4 System Testing

Testing plays a fundamental role in the realization of DREAM; as already mentioned, a test-driven development of the system is imagined and this was one of the reasons why we chose to follow a bottom-up approach in the implementation and integration of the various system components. This approach is, in fact, very effective in bug tracking and allows to carry out subsequent unit tests one on each component implemented and still not integrated. Through testing, the goal is to find as many bugs as possible until the release date of the system; therefore, it is essential to achieve this purpose that the verification and validation phases begin as soon as the development of the system begins and proceed in parallel with its implementation. In addition to unit tests to be performed individually on each component developed, an integration test is required whenever a certain component is integrated with others already implemented and tested. The order in which unit tests and integration tests are carried out follows what is described for the implementation and integration plan. The only components of the system that are not subject to unit tests are the external services (TSDPS and Google Maps) because not being implemented they are already considered reliable; however, they will undergo integration testing once all client and server components have all been implemented, integrated and tested.

When the system is fully integrated, i.e. when all the various integration tests have been completed, it is necessary to proceed with system testing through which it is verified whether the developed version of DREAM satisfies or not the functional and non-functional requirements described in the RASD. For system testing to be functional to its purpose, the testing environment must be extremely similar to the production environment.

In general, system testing can be divided into several types:

1. Functional testing, i.e. the set of tests used to verify whether the system meets the established functional requirements or not.

2. Performance testing, i.e. the set of tests used to verify whether the system meets the non-functional requirements established or not. They are tests that therefore focus on identifying bottlenecks especially in the response time and in the use of the system, by doing so they allow to find hardware / network issues as well as query optimization possibilities that are very useful given the large amount of data managed by the system which implies a heavy interaction with the database.

3. Load testing, i.e. the set of tests aimed at ensuring that a network system can handle an expected volume of traffic, or load limit. The goal of these kind of tests is to prove that a system is capable of handling its load limit, with minimal to acceptable performance degradation. Through the execution of these tests, it will be possible to identify the upper limits of the system components and find bugs related in particular to memory management.

4. Stress test, i.e. the set of tests aimed at ensuring the stability of a certain system as well as its ability to recover from any errors that arise by overloading it past its upper limits, until it breaks. These are tests which deliberately induce a failure scenario in the system under extreme load, so that the risk involved at the breaking points can be analyzed; according to what has already been expressed in the RASD, the system is required to be fault tolerant and therefore it is a priority to ensure that the system recovers gracefully after failure.

When all these tests have been conducted, then system testing can be considered concluded.

6 Effort Spent

Topic	Pietro Valente Hours	Andrea Seghetto Hours	Total Hours
General discussion	together	together	5h
Introduction: Graphic layout	0.5h	0.5h	1h
Introduction: Purpose	0.5h	0h	0.5h
Introduction: Scope	0h	0.5h	0.5h
Architectural Design: Overview	0h	5.5h	5.5h
Architectural Design: Component View	4.5h	0h	4.5h
Architectural Design: Deployment View	0h	4h	4h
Architectural Design: Runtime view	4h	4h	8h
Architectural Design: Component interfaces	3.5h	7h	10.5h
Architectural design: Selected architectural styles and patterns	2h	0h	2h
Architectural design: Other design decisions	1h	0h	1h
Architectural design: DREAM AI Algorithms	6.5h	0h	6.5h
User interface design	1h	0h	1h
Requirements trace-ability	3h	0h	3h
Implementation, integration and test plan	3h	8h	11h
Final revision of the document with changes	4h	4h	8h
Effort tracking	together	together	2h
	40.5h	**40.5h**	**74h**

References

[1] Specification Document: "01. Assignment RDD AY 2021-2022.pdf"

[2] Slides of the lectures

[3] All diagrams have been made with https://app.diagrams.net/

[4] Weather forecast site: https://www.tsdps.telangana.gov.in/aws.jsp

[5] Google maps API site: https://developers.google.com/maps

[6] All the screen and possible interface has been made using Android Studio: https://developer.android.com/studio?gclid=CjwKCAiA-9uNBhBTEiwAN3IlNM0jGYMa-ogbqUe1ZYPbhrkRsx64WxpMt115G2om6klG5qiQ9f0IyhoCE4MQAvD_BwE&gclsrc=aw.ds

[7] Collaborative Filtering: https://realpython.com/build-recommendation-engine-collaborative-filtering/#when-can-collaborative-filtering-be-used

[8] Maurizio Ferrari Dacrema github folder: https://github.com/MaurizioFD/RecSys_Course_AT_PoliMi/tree/master/Recommenders

[9] Header "If-Modified-Since": https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/If-Modified-Since

