POLITECNICO
MILANO 1863

Design Document

Maccarrone Salvatore, Pellizzer Massimiliano, Prati Andrea

January 5, 2022

Contents

Chapter 1

Introduction

Agriculture plays a pivotal role in India's economy. Climate change continues to be a real and potent threat to the agriculture sector, which will impact everything from productivity to livelihoods across food and farm systems and is predicted to result in a 4%-26% loss in net farm income towards the end of the century. This calls for a revamp of the entire mechanism that brings food from farms to our plates. The COVID-19 pandemic has greatly highlighted the massive disruption caused in food supply chains exposing the vulnerabilities of marginalized communities, small holder farmers and the importance of building resilient food systems. It has become even more important now that we develop and adopt innovative methodologies and technologies that can help bolster countries against food supply shocks and challenges.

1.1 Purpose

Telangana's government wants to design, develop and demonstrate anticipatory governance models for food systems using digital public goods and community-centric approaches to strengthen data-driven policy making in the state. This will require the involvement of multiple stakeholders, from normal citizens to policy makers, farmers, market analysts, agronomists and so on.

In the first place, Telangana wants to partner with IT providers with the aim of acquiring and combining:
Data concerning meteorological short-term and long-term forecasts. Telangana already collects and makes available such data (see https://www.tsdps.telangana.gov.in/aws.jsp).

- Information provided by the farmers about their production (types of products, produced amount per product).
- Information obtained by the water irrigation system concerning the amount of water used by each farmer.
- Information obtained by sensors deployed on the territory and measuring the humidity of soil.
- Information obtained by the governmental agronomists who periodically visit the farms in their areas.

Acquiring and combining such data, DREAM will support the work of three types of actors: policy makers, farmers, and agronomists. In the following we describe the goals summed up.

1.1.1 Goals

G1 - Data-Driven approach

DREAM will be able to fetch data from different databases. Then, DREAM will be able to collect, analyze and perform inference on this data in order to extract new knowledge that will be at the service of different *DREAM*'s users.

G2 - Help policy makers to implement better policies which will make Telangana's agriculture better

DREAM will allow policy makers to see real time data regarding Telangana and Telangana's farms. In particular they will be able to see which farms are performing well and which farms are performing poorly. Moreover, they will be able to understand which steering initiatives are working and which not.

G3 - Help farmers to become weather resilient and increase their production

DREAM will allow farmers to see statistics regarding their farms. Moreover, *DREAM* will provide to them personalized suggestions based on the data regarding their farm and on the data regarding farms that are doing particularly well in similar environments.

G4 - Facilitate and make systematic the communication between farmers and agronomists

DREAM will make communications between agronomists and farmers easy and stable. In fact, *DREAM* will allow farmers to chat with agronomists in just a few clicks. Moreover they will be able to request agronomists to visit them when they are facing complex problems. Agronomists, instead, will be able to access real time data regarding farms they are responsible for, and therefore they will be able to see which farms are performing well and which farms are performing poorly. Based on this knowledge, agronomists will be able to decide how much and when to visit each farm, and to send periodical reports to policy makers regarding how the farms (they are responsible for) are doing.

G5 - Create a network of farmers

Through *DREAM*, farmers will be able to communicate between each other in order to exchange best practices and advice.

1.2 Scope

The problem that Telangana is facing concerns the world of agriculture, and it has a very important weight on the lives of many people who, due to various impediments mentioned in the previous section, need a platform that can support them and allow those who create support initiatives to do the work in the most scientific way possible, keeping track of how is all going on without losing information.

To reach this goal, called "data-driven approach" it was necessary to think at an entire platform. In particular, this platform can be seen as divided into three categories:

- Data: users must be able to easily interact with the data to which they have the right to access. They are offered different views and aggregations with respect to this data. The available data can include weather forecasts, the state and composition of the land of an area of Telangana, the water irrigation system (the amount of water used), and the initiatives taken by other farmers who have distinguished themselves for being productive considering the conditions in which they were found (positive and negative deviance). Other types of data are described in the remainder of the document.

- Community: users are able to communicate via two communication channels within the system, one is the Forum and the other is the Chat System that will allow Farmers and Agronomists to talk to each other to exchange advice and clarifications.
- Management: users can organize physical visits between Agronomists and Farmers directly in the app.

The processing of data and their analysis was not the focus for the drafting of this document. In fact, the first goal was to think about how to develop the platform as a whole. The choice on which criteria to calculate the Farmer's performance (i.e. their deviance, positive or negative) will be made thanks to the collaboration with expert agronomists who will provide all the knowledge to be able to create and study intelligent models using special tools such as, TensorFlow, a framework for processing data.
Moreover, the system will interface itself with different databases maintained and managed by third parties. How the data are inserted or managed in those databases is not one of the concern of the system described in this document.

1.2.1 Phenomena

World Phenomena

$W01$ The farmer weighs the crop

$W02$ Physical event that the farmer considers important to report happens

$W03$ The farmer faces a problem concerning their farm

$W04$ The agronomist misses an appointment

$W05$ The agronomist goes to the farmer

$W06$ The agronomist takes measures

$W07$ The agronomist assists the farmer while he is in visit to their farm

$W08$ The farmer open a chat

Shared Phenomena

$S01$ Registration of the user (World Controlled)

$S02$ Login of the user (World Controlled)

$S03$ The user queries the system (World Controlled)

$S04$ The system shows data to the user (Machine Controlled)

$S05$ The user queries the system to retrieve relevant data (World Controlled)

$S06$ The system shows agronomists the list of farmers they have to manage (Machine Controlled)

$S07$ The farmer inserts data concerning their production (World Controlled)

$S08$ The system confirms the success of an operation to the user (Machine Controlled)

$S09$ The farmer open a ticket in order to have a chat with an agronomist (World Controlled)

$S10$ The system shows an agronomist the list of their currently open chats (Machine Controlled)

$S11$ The system shows agronomists the list of chat requests to be managed (Machine Controlled)

$S12$ The agronomist accepts a chat request (World Controlled)

$S13$ The agronomist closes a chat (World Controlled)

$S14$ The farmer interacts with the system's forum (World Controlled)

$S15$ The agronomist schedules an appointment in order to visit a farm (World Controlled)

$S16$ The agronomist cancels an appointment from the schedule (World Controlled)

$S17$ The agronomist confirms the scheduled appointment done during the day (World Controlled)

$S18$ The agronomist inserts a report and other data regarding the appointment done during the day (World Controlled)

$S20$ The agronomist get notified

$S21$ The farmer get notified

Machine Phenomena

$M01$ The system periodically fetches data from its sources

$M02$ The system aggregates and filters data

$M03$ The system validates data and stores it

$M04$ The system updates the agronomist's schedule

$M05$ The system check the references of the agronomist

$M06$ The system checks credentials of the user

$M07$ The system generates the list of farmer associated to each agronomist

$M08$ The system orders the list of farmers associated to each agronomist by priority

$M09$ The system keeps updated the priority of farmers inside their list

$M10$ The system removes a chat request from the list and adds a new entry in the list of open chats managed by an agronomists

$M11$ The system removes the chat from the list of open chats of the agronomist that taken it into account

$M12$ The system synchronizes lists shared between agronomists working on the same area

$M13$ The system keeps track of the appointment missed by agronomists

$M14$ The system generates personalised suggestions based on historical data to be delivered to the farmer

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

Agronomist's schedule → The schedule of an agronomist is a list of planned activities or things to be done showing the times or dates when they are intended to happen or be done. In particular, in this case, the activities are the planned visits to the various farms.

Area → The word "area" indicates a disjoint partition of Telangana. In fact, Telengana is partitioned in several disjoint areas, and to each area are associated both farmers and agronomists, in such a way that:

- Each area can have one or more farmers and one or more agronomists
- Each farmer is associated to one and only one area
- Each agronomist is associated to one and only one area

Chat → A chat is a virtual space that, after it has opened, allows one agronomist and one farmer to discuss with each other.

Data Summary → A data summary is an aggregation of data fetched by various databases, which gives a certain data view of a subject (e.g. a farm or an area).

Data View → A data view is a selection of data fetched by the various databases, regarding a specific aspect (e.g. humidity of soil, water irrigation data) of the subject (e.g. a farm or an area of Telangana).

Priority → The term "priority", when used referring to the list of farms, indicates that the list of farms is sorted taking into account:

- The number of times the farm has already been visited during the year
- How well or bad the farm is performing
- Possible requests of help done by the farmer

These metrics define what is referred to as priority.

Relevant data → The term "relevant data" indicates data regarding:

- Data concerning meteorological short-term and long-term forecasts
- Information provided by the farmers about their production
- Information obtained by the water irrigation system concerning the amount of water used by each farmer
- Information obtained by sensors deployed on the territory and measuring the humidity of soil
- Information obtained by the governmental agronomists who periodically visit the farms in their areas

Report → When an agronomist visits a farm they must write (and insert into the system) a summary of what they have suggested to the farmer, which problems they have detected, and everything they think could be relevant in order to understand why the farmer is performing well or why the farmer is performing poorly. This summary is indicated by the word "report". The purpose of the report is to keep track of the history of every farm.

Ticket (or Chat Request) → Whenever a farmer wants to have a chat with an agronomist they have to open a ticket. A ticket, thus, is a request to have a chat.

Open a ticket → Open a ticket indicates the action, performed by the farmer, of asking an agronomist to have a chat.

Managed ticket → A managed ticket is a chat request accepted by an agronomist.

Not manged ticket → A ticket that is not managed is a chat request not accepted by any agronomist yet.

Users → The users of the system are: farmers, agronomists and policy makers. Whenever the word "users" is used, it indicates both agronomists, farmers and policy makers.

Visit → In order to help farmers, agronomists can reach the farm of the farmer physically. A visit, therefore, is the action, performed by an agronomist, of physically reaching a farmer's farm.

Booking a visit → To book a visit means reaching an agreement between a farmer and an agronomist on when the agronomist should visit the farm.

Booked visit → A booked visit is a visit that has been scheduled by the agronomist in accordance with the farmer

1.3.2 Acronyms

API → Application Programming Interface, it indicates on demand procedure which supply a specific task.

BPMN → Business Process Model and Notation, it is a graphical representation for specifying business processes in a business process model.

DBMS → Data Base Management System, it is an interface between the end user and the database, simultaneously managing the data, the database engine, and the database schema in order to facilitate the organization and manipulation of data.

DREAM → Data-Drive Predictive Farming in Telengana, is the name software to be developed, described in this document. IOT

1.4 Revision History

1.5 Reference Documents

Multiple Tier Architecture → https://www.ibm.com/cloud/learn/three-tier-architecture

Big Data Architectures → https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/

Apache Spark → https://spark.apache.org/

1.6 Document Structure

Chapter 2

Architectural Design

2.1 Overview: High-Level Components and their interaction

The system to develop is a distributed application which follows a 4-tier client-server architecture. In particular, the application will be divided in the following tiers:

1. Presentation Tier
2. Application Tier
3. Data Tier
4. Big-Data Tier

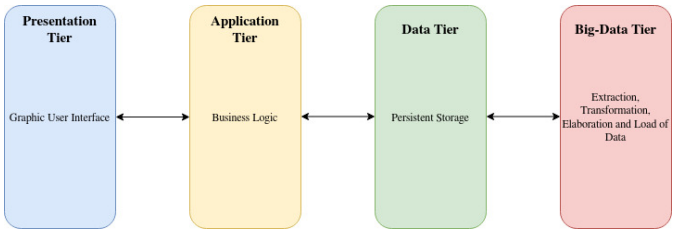

Figure 2.1: Four-Tier Application (High-Level View)

The presentation tier is where the user interface and the communication layer of the application are implemented and where the user interacts with the application. Its main purpose is to display information to the user and collect information from the user. This top-level tier will run both on thin clients as well as thick clients, in fact, the presentation tier software can run both on a web browser, as well as mobile application.

The application tier, also known as the logic tier or middle tier, is where information collected in the presentation tier is processed (sometimes against other information in the data tier) using business logic, which means a specific set of business rules. The application tier can also add, delete or modify data in the data tier, in fact, this tier is able to communicate with the data tier by using API calls.

The data tier is where the information processed by the application is stored and managed. In particular this tier will contain the DBMS used by the application in order to store and retrieve data.

The big-data tier is the tier responsible to automatically fetch data both from, elaborate and store data into the internal database in order to extract valuable knowledge from raw data. In particular this tier will be responsible for automatically fetching data both from the internal database as well as from the third parties' databases (through known API), elaborate those data, and store the results into the internal database.

Below is shown the architecture adopted in more detail:

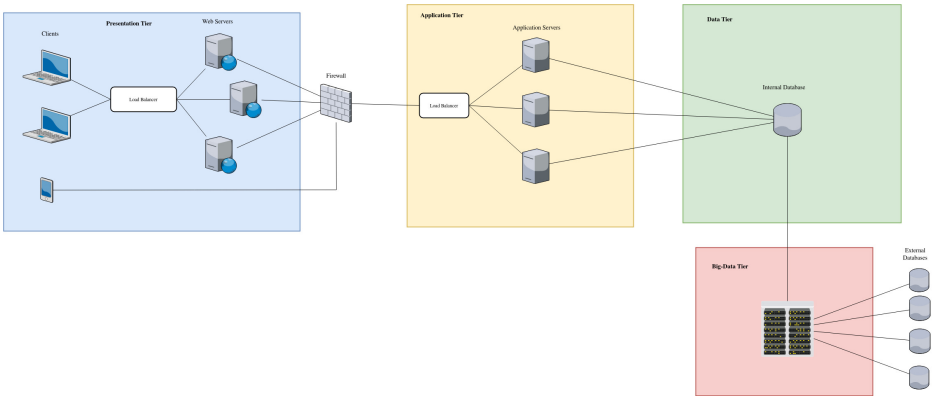

Figure 2.2: Four-Tier Application (Low-Level View)

It is possible to notice that the system is supposed to be accessed through both a web interface and a mobile application, and this is valid for all kinds of users. In particular, those users which use a web browser will communicate with a web server (which, in turn, interfaces itself with the application tier), while, in case of using the mobile application, the core of the software installed on the user's device will interface with the business layer's APIs, which send and receive all the information, in order to work properly, directly to the application servers.

The presentation tier and the application tier is divided by a firewall, for security purposes.

The communications between the presentation tier and the application tier are REST (Representational State Transfer) compliant, which means that:

- Interactions are client-server and stateless. This means that the history is not accumulated with time, giving an advantage in term of scalability. Eventually a caching service can be used in order to reduce latency in case the clients send multiple time the same request, avoiding the server to execute multiple time the same functions.
- The data within a response to a request must be implicitly or explicitly labeled as cacheable or non-cacheable. The ability of caching data is key to provide scalability.
- Each component cannot "see" beyond the immediate layer which it is interacting with.
- Components expose a uniform interface.

The application tier, which contains the application's logic, will use an ORM (Object-Relational Mapping) programming technique in order to interface with the DBMS exploiting the advantages of the object-oriented paradigm.

It is possible to see that both the web server and the application server can be replicated, in order to manage the high loads without affecting the performance.

The data tier, as said before contains the internal database of the system. The internal database will be replicated and distributed, and it will possibly hosted in the cloud.

The big-data tier is responsible for fetching data both from internal and external databases, make computations on them, and store the results in the proprietary database. In fact, this tier is responsible for analyzing data, extracting statistics from them, and making inference on them. Given the huge amount of data to process this tier must be hosted in cloud, in order to be scalable and to able to manage higher and higher loads of data. More on this tier will be explained in a dedicated section.

2.1.1 Class-diagram

In order to provide a better overview about the system and their components here is the UML diagram of the system which will be useful in order to get a first view of the logic and the interactions between the classes. We structured the system with three type of users, as requested by the assignment, they are all an extension of the class **User**, which is the one that can interact with the *Data layer*, thus every user has the possibility to access at some data, the type of access depends on the type of instance the user is.
The so called *Data layer* has a general handler called **Data handler** that's the one that include a **Data viewer**, a **Data collector** and list of **Data**, here a short description of these classes:

- **Data viewer**: it's the class which has the responsibility of providing all the methods to proper visualize the data. Here there is the access with the Machine Learning component, in order to see this it's better to look at the Design Document.
- **Data collector**: it's the class that allow to collect data and make all the processing needed to store them properly into the DB.
- **Data**: it's the class used to consider the Data entity, thus every data which comes from or goes to the DB it's a **Data** object.

The **Data Handler** interact also with the external persistent storage, which is better specified in the Design Document, and the **Community Handler**, which is the class that has the responsibility of managing the **Chat** and the **Forum**, in order to store info about the *chats* and the forum *threads*.

Each **Farmer** and each **Agronomist** has a **Calendar** and a **Community Handler**. The big difference we can notice between the two type of **User**, the **Agronomist** and the **Farmer**, is that the **Farmer** is composed of a **Farm** and each **Farm** is contained in just one **Area** and an **Area** can contains many *farms*, while an **Agronomist** is bound to just one **Area** while each **Area** can have more Agronomist bound to itself.
Notice that the **Area** class is an extension of a **GeoSpatialPosition**.

2.2 Component View

Questo è il *component diagram* del sistema DREAM nel quale vengono mostrati diversi componenti, ognuno dei quali va considerato con un certo livello di astrazione. Il *component diagram* è stato pensato tenendo conto anche del *class diagram* presentato e descritto alla sezione 2.1.1 di questo documento. Essendoci dunque un certo mapping tra i due diagrammi alcune cose potrebbero risultare ridondanti. Da notare che il mapping non è stato effettuato considerando la regola "class = component", infatti non sarebbe stato possibile separare le componenti software nelle varie componenti hardware, inoltre questo *component diagram* presenta un livello di astrazione per certi versi più alto anche se vuole fornire più dettagli

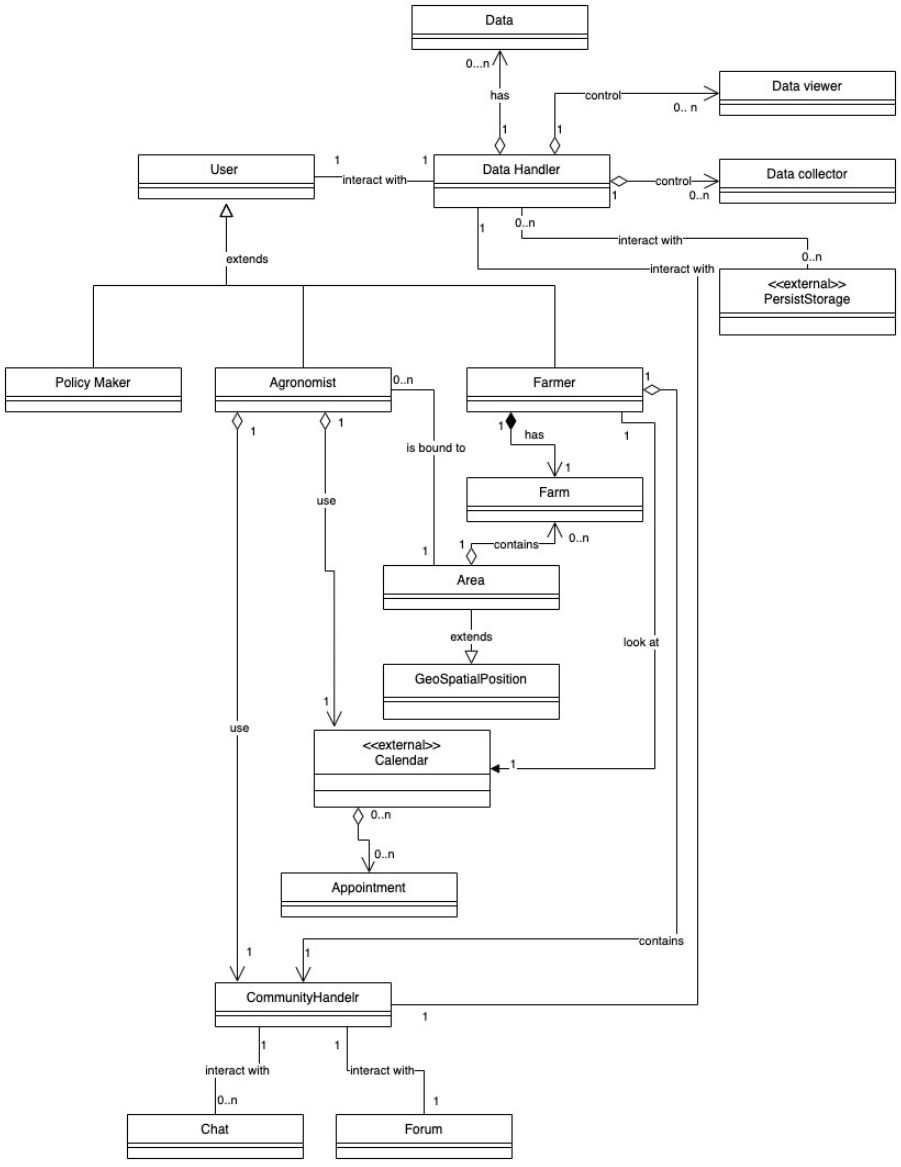

Figure 2.3: Class diagram

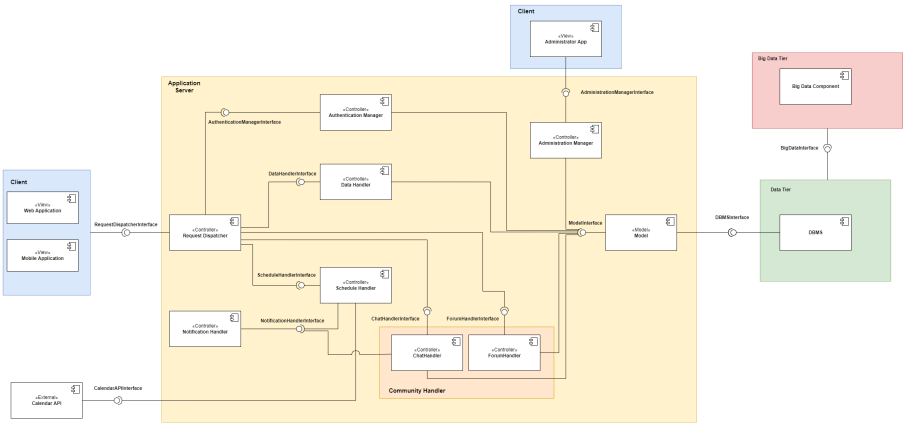

Figure 2.4: Component Diagram

riguardanti la struttura del sistema e le interazioni delle componenti per mezzo delle interfacce.
Iniziamo a descrivere questo diagramma a partire dalla parte più a destra dell'immagine. Le due componenti **User** e **ITadim** rappresentano rispettivamente gli utenti presenti nell'applicazione (policy maker, agronomist e farmer) e l'adim che si occuperà di manutenere il sistema. Entrambe queste componenti comunicano con l'**ApplicationView**, quella componente che ha il ruolo di astrarre la parte visiva del sistema, in particolare questa rappresenta sia la *web application* che la *mobile application*.
Dall'ApplicationView vi sono tre interacce che portano ai controller dei tre sottosistemi di DREAM descritti nella sezione 1.2 di questo documento: *Management*, *Community* and *Data*. In particolare i tre sottosistemi sono rappresentati da tre handler, i quali avranno il compito di gestire e indirizzare le richieste diretta alla parte più interna dei diversi sottosistemi. Qui di segutio parliamo in maniera più dettagliata dei tre handler, da notare che questi si trovano all'interno del **Server** del sistema.

- **CommunityHandler**: a questa componente arriveranno tutte le richieste riguardanti le interazioni con la parte di community di DREAM. Infatti questa componente per funzionare richiede l'esistenza di un'interfaccia che il componente **CommunityModel** mette a disposizione.

- **ManagementHandler**: questa componente gesisce tutte le richieste riguardanti le interazione con la parte di management di DREAM, quindi tutta la gestione delle farm, delle aree e così via. Questo component si attacca all'interfaccia messa a disposizione del **ManagementModel**.

- **DataHandler**: questo component gestirà le richieste provenienti dall'**ApplicationView**, dal **CommunityHandler** e dal **ManagementHandler**.
  Dei tre handler questo è il più complicato, infatti è necessario specificare i principali sottocomponenti. Essendo DREAM nata sotto richiesta di essere un'applicazione *Data Driven*, un solo handler per questo aspetto non può bastare e va quantomento suddiviso in quattro subcomponents:

  - **UserAuth**: questo component si occupa di gestire tutta la parte di registrazione e login degli utenti
  - **UserDispatcher**: è il component che si occupa di gestire tutta la *logica di permessi* riguardanti la visualizzazione dei dati nei confronti del tipo di user che fa richiesta per la visualizzazione di qualcosa. Si propone quindi di usarlo come livello di sicurezza aggiuntivo per evitare richieste illecite da parte degli utenti.

- **ModelDataController**: qui viene racchiusa tutta la logica di gestione delle richieste funzionali (quindi tutto ciò che riguarda metodi funzionali) provenienti dagli altri due Handler
- **AppointmentController**: qui vengono gestite le richieste per appuntamenti, sia per quanto riguarda la creazione che la modifica e la cancellazione degli appuntamenti. Per la gestione logica di questo aspetto vi è un component chiamato **Appointment** dedicato, esso è collegato a un'interfaccia esterna messa a disposizione dal component **Calendar**, il quale non è altro che una componente, appunto, esterna a DREAM (vedi GoogleCalendar, OutlookCalendar, etc).

Le prime due componenti sopra descritte, per poter funzionare, sono anch'esse connesse alla componente **DataHandler** che verrà quindi utilizzata come punto di accesso al **PersistentStorage**, componente che si occupa di contenere tutta la logica sulla persistenza dei dati, quindi la comunicazione con il DBMS e quello che abbiamo definito come **BigDataTier**. Da notare che il **PersistentStorage** si trova all'interno del **DataTier**.
Le due componenti interne al **CommunityModel** sono **Chat** e **Forum** nei quali viene racchiusa tutta la logica riguardo, appunto, il funzionamento delle chat e del forum.
Il **DataHandler** è collegato tramite due interfacce, da lui stesso messe a disposizione, alle due componenti **DataCollector** e **DataViewer**, il primo gesisce la logica di pulizia e processamento dei dati prima che vengano rimandati ancora una volta al **DataHandler** il quale invece si occuperebbe di mandare tutto al **DataTier**, mentre il secondo, il **DataViewer** si occupa di gesitre tutte le richieste di visualizzazione di *data summary* di qualisasi genere, infatti questo component, ricevute le richieste, chiederà al **DataHandeler** i dati che da prelevare dal **DataTier** in quanto necessari alla creazione del *data summary*. Oltreutto il **DataViewer** preparerà i dati prelevati dal **PersistentStorage** per essere leggibili dal sistema e quindi infine dall'**ApplicationView**.
Resta solo da descrivere meglio il component **ManagementModel** e i suoi subcomponent. Questo compoent contiene:

- **Farm**: si occupa della gestione logica delle farm. È dipendente dall'**area**, infatti senza l'esistenza di questa la Farm non potrebbe funzionare.
- **Area** and **GeoSpatialPosition**: Nella prima vengono racchiuse tutte le informazioni riguardanti l'area, quindi tutte le farm a lei associate e i rispettivi farmer, e tutti gli agronomi presenti sul territorio. Ad ogni farm associa una GeoSpatialPosition, ovvero una posizione geografica all'interno della regione di *Telangana*.

2.3 Deployment View

2.4 Big-Data Tier

The big-data tier was introduced in order to handle the ingestion, processing, and analysis of data that is too large or complex for traditional database systems. This tier, in fact, is responsible for periodically fetching data from the different databases (both the internal one as well as the external ones), transform and elaborate the data in order to extract valuable knowledge from it, and store the results of different computations inside the internal database.

2.4.1 Components of the Big Data Tier

The following diagram shows the logical components that typically fit into a big data tier.

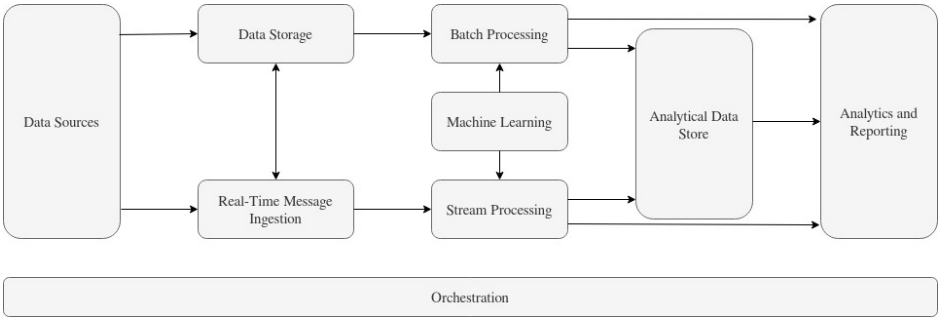

Figure 2.5: Generic Big Data Architecture Components

Here there is a brief description of each component:

- *Data Sources* They are the various sources from witch the system extracts data. They could be relational databases, static files produced by the application, as well as real-time data sources, such as IoT devices.
- *Data Storage* This component is a distributed file store that can hold high volumes of large files in various formats. It is used in order to store data for batch processing operations.
- *Batch Processing* Often a big data solution must process data files using long-running batch jobs to filter, aggregate, and otherwise prepare the data for analysis. The batch processing component, therefore, is responsible for jobs involve reading source files, processing them, and writing the output to new files.
- *Real-Time Message Ingestion* If the system includes real-time sources, the architecture must include a way to capture and store real-time messages for stream processing. Thus, this component buffers messages, supports scale-out processing, reliable delivery, and other message queuing semantics.
- *Stream Processing* After capturing real time messages, the system uses the stream processing component in order to filter, aggregate, and prepare data for analysis. Then, it writes the processes data stream to an output sink.
- *Analytical Data Store* Before analytical tools start their execution they have to fetch data in a structured format. Here comes into play the analytical data store, a component in which the system stores processed data in a structured format.
- *Analysis and reporting* This component is the one responsible for providing insights into the data through analysis and reporting.

It is important to notice that the system may not implement all the aforementioned components in the first version of the software. However, as the application grow both in terms of users and functionallity, all of them may be required in the future.

2.4.2 Apache Spark

Apache Spark is an open source cluster computing framework for both batch data processing and real-time data processing. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. It is designed to cover a wide range of workloads such as batch applications, iterative algorithms, interactive queries, and streaming.

Apache Spark is made up of different modules, which fully cover the functions performed

by the components listed in the previous section. In the following diagram are shown the most used modules of Apache Spark:

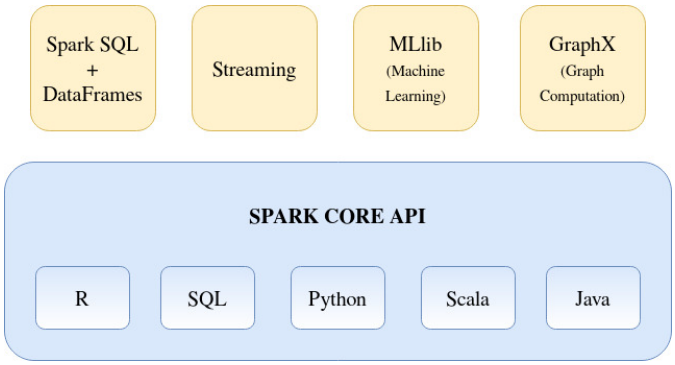

Figure 2.6: Apache Spark Modules

Here there is a brief description of each module:

- **Spark Core**
  Spark Core is the base engine for large-scale parallel and distributed data processing. It is responsible for memory management and fault recovery, scheduling, distributing and monitoring jobs on a cluster and interacting with storage systems. Spark Core API is available in R, SQL, Python, Scala and Java.

- **Spark Streaming**
  Spark Streaming is the component of Spark which is used to process real-time streaming data. Thus, it is a useful addition to the core Spark API. It enables high-throughput and fault-tolerant stream processing of live data streams.

- **Spark SQL and DataFrames**
  Spark SQL is a module of Spark which integrates relational processing with Spark's functional programming API. This module makes Spark ideal for extracting data from databases as well as for storing new data inside them.

- **GraphX**
  GraphX is the Spark API for graphs and graph-parallel computation.

- **MLlib (Machine Learning)** MLlib stands for Machine Learning Library. Spark MLlib is used to perform machine learning in Apache Spark. This module can be used in order to extract valuable knowledge from our data assets.

Given the fact that Apache Spark is an open-source framework (it is under the "Apache Licence 2.0", that is compliant with the Digital Public Good Standards), and given that it's modules implement all the functionality needed by our data-driven software, it is a reasonable choice to adopt Apache Spark framework in order to implement the big-data tier.

2.5 Runtime View

The following sequence diagrams show how the components interact with each other whilst the functionalities are executed. It has been assumed that the users interact with the application through a mobile application. Nonetheless, the system operates in a very similar way from the web application. The only change is that the user interacts with the WebApplication component instead of the MobileApplication component.

2.5.1 Login

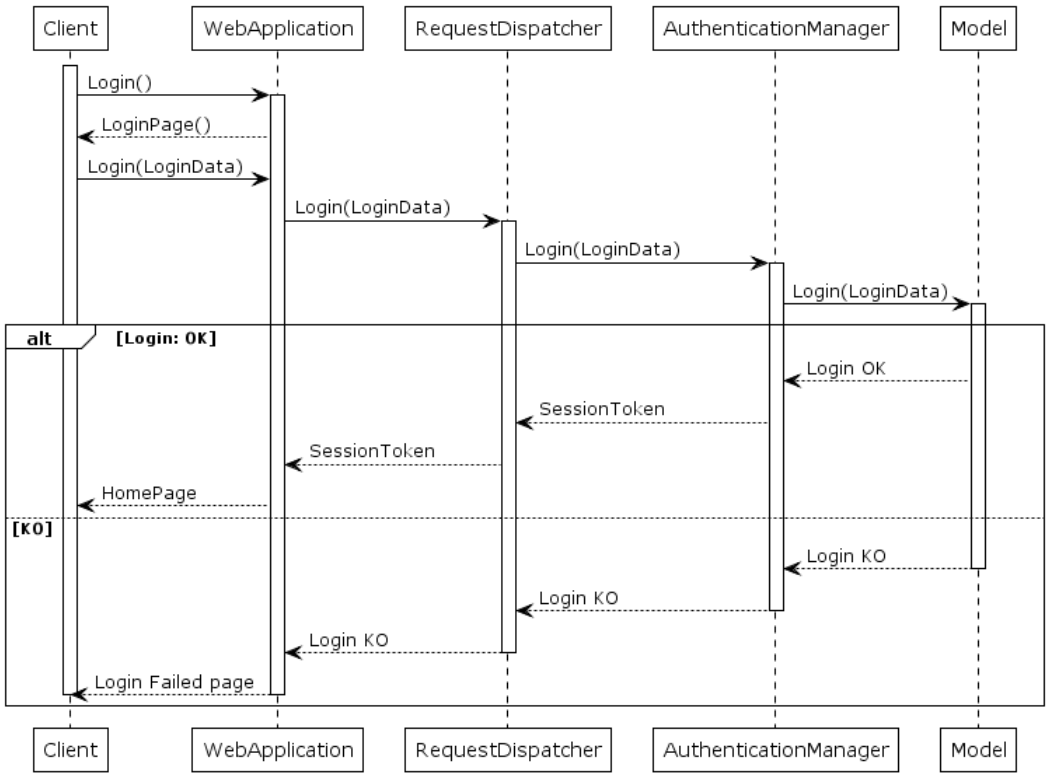

Figure 2.7: Runtime view: Login

This diagram represents the interactions that happen when a user of the software to be wants to login on the web application. The web application shows a login page with a login form. Once the login form is filled, the web application sends it to the **RequestDispatcher** that routes the request to the **AuthenticationManager**. The AuthenticationManager knows how to contact the model component which interacts with the DBMS. If the login is successful a session token is set, otherwise the web application simply returns an error page. In the following sequence diagrams, the client is supposed to be correctly logged into the web application.

2.5.2 Sign up

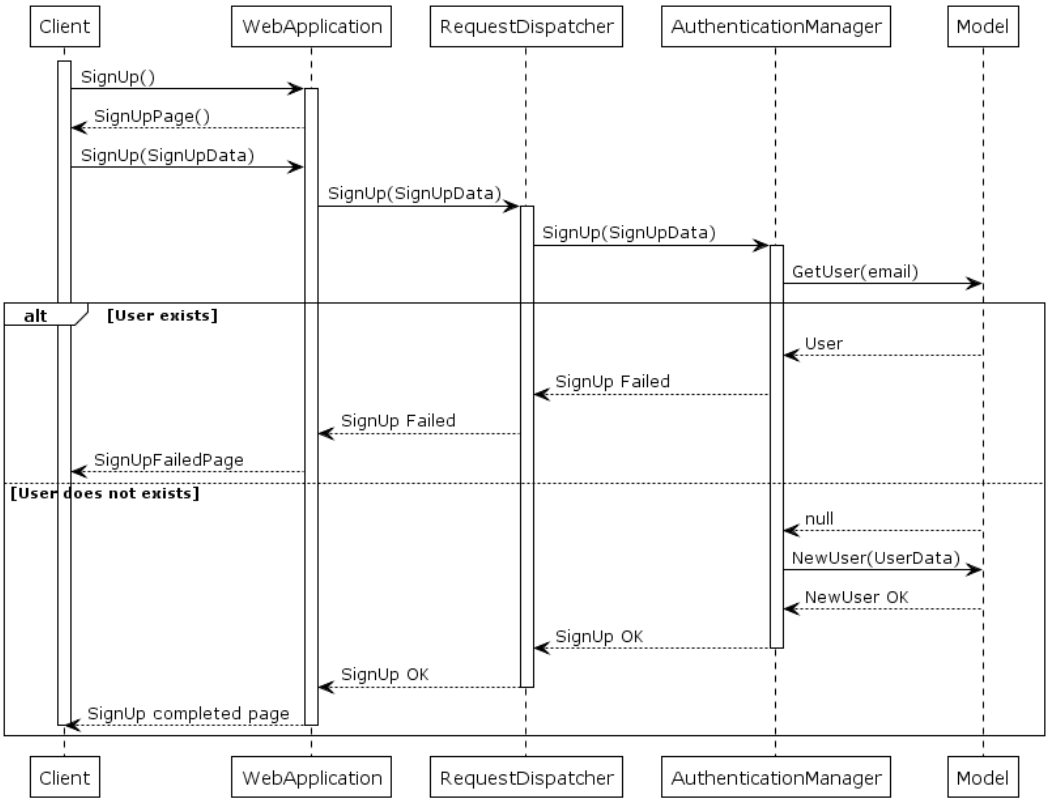

Figure 2.8: Runtime view: Sign up

This diagram represents the interactions that happen when a user of the software to be wants to sign up on the web application. The web application shows a sign up page with the appropriate form. Once the form is filled, the web application sends it to the **RequestDispatcher** that routes the request to the **AuthenticationManager**. The AuthenticationManager knows how to contact the model component which interacts with the DBMS. In particular, the **AuthenticationManager** checks if the email provided by the client is already present in the system: if so the sign up request is rejected, the web application displays an error page. Otherwise, the **AuthenticationManager** asks the Model to create a new user, and when the operation is completed, it tells the **RequestDispatcher** that the request was completed. The web application then shows a page showing that the signup is completed.

2.5.3 Farmer asks help through the chat

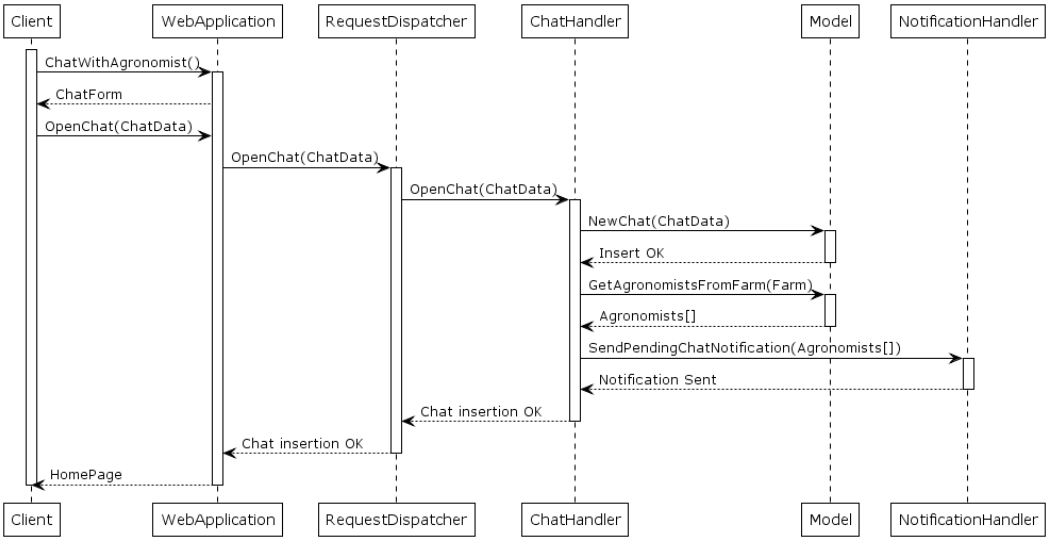

Figure 2.9: Runtime view: Farmer asks help through the chat

This sequence diagram shows what happens when an already logged Farmer wants to open a chat. The web application takes in input the request from the farmer and sends it to **RequestDispatcher**. It knows that this request has to be processed by the **ChatHandler**, so the request routed to this component. The **ChatHandler** has two tasks: it needs to insert the chat into the DBMS, and to notify the agronomists that belong to the same area of the farmer. To complete the latter task, the **ChatHandler** must retrieve the agronomists that need to be notified about the new chat. When those tasks are completed the **ChatHandler** returns to **RequestDispatcher** with a response telling that the operation was successful. The **web application** then displays the home page again.

2.5.4 Farmer creates a new thread in the forum

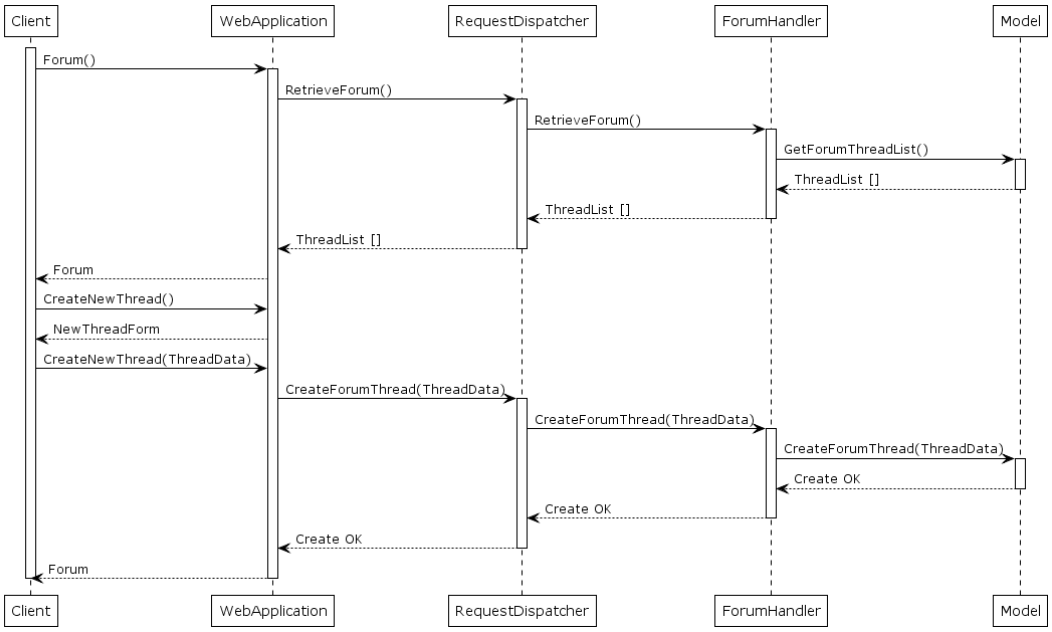

Figure 2.10: Runtime view: Farmer creates a new thread in the forum

This sequence diagram shows what happens when an already logged **Farmer** wants to create a new thread in the forum.

When the Farmer wants to see the forum the web application needs to retrieve it from the DBMS. The request is routed from the **RequestDispatcher** to the **ForumHandler** that contacts the model to obtain what is needed to display the forum. When the web application shows the forum the farmer selects the option to create a new thread. So a form is displayed by the web application to the user, in order to collect the data about the new thread. The web application sends to the RequestDispatcher the request to create a new thread, with the associated data coming from the farmer input. The **ForumHandler** deals with this request by asking the model to create a new thread into the DBMS. If the insert on the DBMS is ok the **ForumHandler** returns to the **RequestDispatcher** that will return to the web application. The farmer is then showed the updated forum page.

2.5.5 Farmer inserts production data about their farm

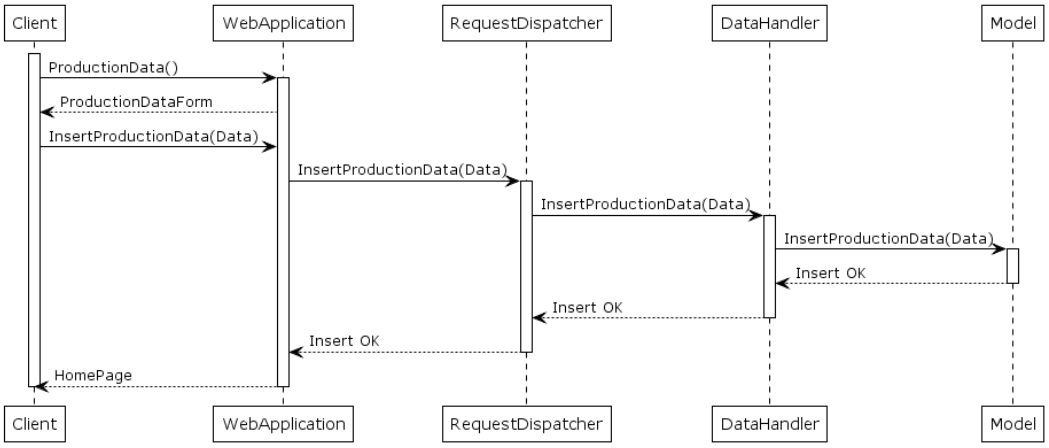

Figure 2.11: Runtime view: Farmer inserts production data about their farm

This sequence diagram shows what happens when an already logged **Farmer** wants to insert production data about their farm.

When the form is filled by the farm, the web application sends to the **RequestDispatcher** a request to insert production data. The request is routed from the **RequestDispatcher** to the **DataHandler**. The request is processed and then the **DataHandler** interacts with the **Model** to persist the data. If the insert on the DBMS is ok the **DataHandler** returns to the **RequestDispatcher** that will return to the web application. The farmer is then showed the home page.

2.5.6 Agronomist takes charge of a chat request

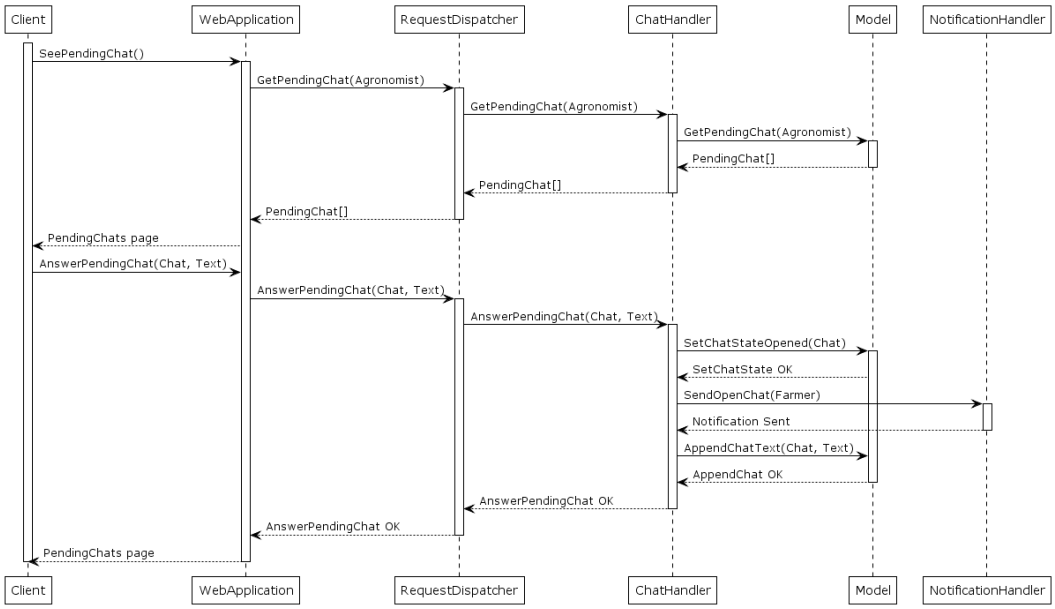

Figure 2.12: Agronomist takes charge of a chat request

This sequence diagram shows what happens when an already logged **Agronomist** wants to take charge of a chat request.

When the Agronomist asks to see the chat request in pending state, the **web application** asks the **RequestDispatcher** that routes the request to the **ChatHandler**. It retrieves the pending chats associated to the area of the Agronomist. When the Agronomist answer the chat, the request that was routed again to the **ChatHandler** is processed. In fact, the **ChatHandler** updates the state of the chat (from pending to opened) and then asks the **NotificationHandler** to notify the Farmer that their chat has been opened. Then, if provided, the answer coming from the agronomist is appended to the chat. If the both the operation on the DBMS are successful, and the NotificationHandler confirms that the notification has been sent, the **ChatHandler** returns a positive answer to the **RequestDispatcher**. The web application then shows the updated pending chats page.

2.5.7 Agronomist schedules a visit to a farm

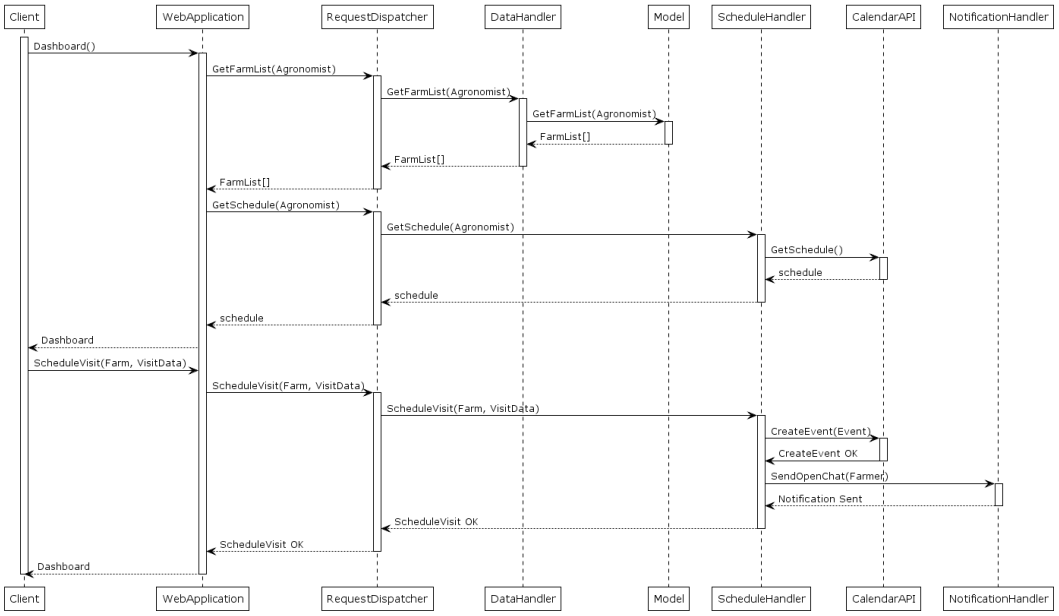

Figure 2.13: Agronomist schedules a visit to a farm

This sequence diagram shows what happens when an already logged **Agronomist** wants to schedule a visit to a farm.

The Agronomist want to see the dashboard. The web application retrieves it in two steps.
First, going through the RequestDispatcher, gets the farm list from the **DataHandler**.
Second, always going through the RequestDispatcher, gets the schedule from the **ScheduleHandler**. This component is the one that interacts with the external **Calendar API**.
Once the dashboard is displayed, the Agronomist can select a farm, and schedule a visit selecting time and date of the visit.
The request is routed to the **ScheduleHandler**: it creates the event in the calendar using the external **Calendar API** and then asks the **NotificationHandler** to notify the farmer that a visit to their farm has been scheduled.

2.5.8 Agronomist confirms a scheduled visit

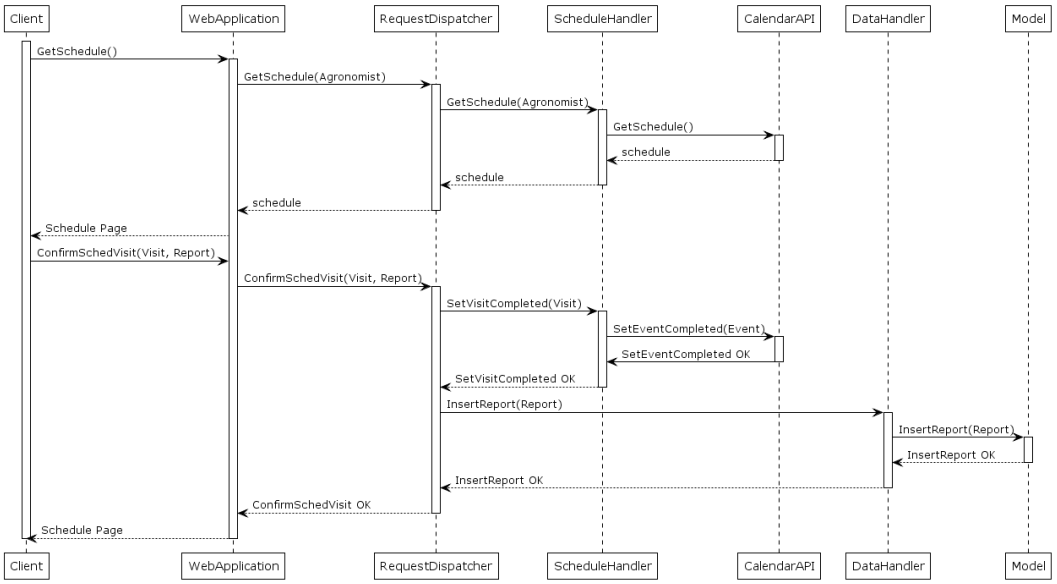

Figure 2.14: Agronomist confirms a scheduled visit

This sequence diagram shows what happens when an already logged **Agronomist** wants to confirm a scheduled visit.
As already clarified in RASD document, this action allows the Agronomist to confirm that he actually visited a farm, that was planned in their schedule. This means also that a report must be provided.

The web application retrieves the schedule, from the **ScheduleHandler** that interacts with the external **Calendar API**.
Then the Agronomist can select a past visit, and requests to confirm it. Here the Agronomist must insert the report of the visit.
The request, is sent to the **RequestDispatcher**. Here the dispatcher has to interacts with two side of the application logic, the one associated to the scheduling of the visits (the **ScheduleHandler**), and the one associated with the data (the **DataHandler**).
First, it contacts the **ScheduleHandler** to change the state of the visit to "completed".
Then it has to store persist the report associated with the visit in the internal DBMS, through the **DataHandler**.

2.5.9 Agronomist cancels a scheduled visit

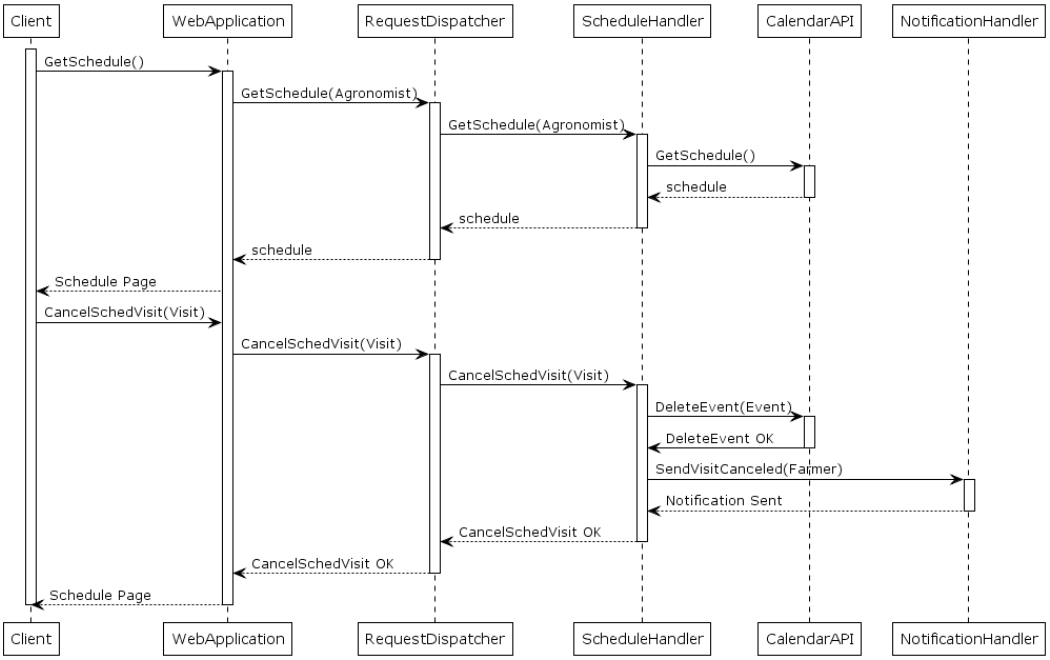

Figure 2.15: Agronomist cancels a scheduled visit

This sequence diagram shows what happens when an already logged **Agronomist** wants to cancel a scheduled visit.

Right after the retrieving of the schedule, the Agronomist selects a visit to cancel it. The request goes to the **ScheduleHandler** that interacts with the external **Calendar API** to delete the event.
Furthermore, the **ScheduleHandler** knows that, dealing with a request to cancel a visit, it has to tell the farmer that the visit is canceled. This is why it contacts the **Notification-Handler**, that returns when the notification is sent.
At this point, the **ScheduleHandler** can confirm to the web application (through the RequestDispatcher) that the operation went well.

2.6 Component Interfaces

The following diagram shows the methods mentioned in the sequence diagrams of the previous section. Furthermore, it clarifies to which interface belongs each method and how the components and interfaces interact with each other in the system. The names of the methods defined in the interfaces are self-explanatory and are not meant to be followed precisely by the developers. They are just a simplified representation of what the different components can offer and how they interact with each other. This artifact should clarify how the system structure is organized to the stakeholders as well as being a simple guideline for the developers.

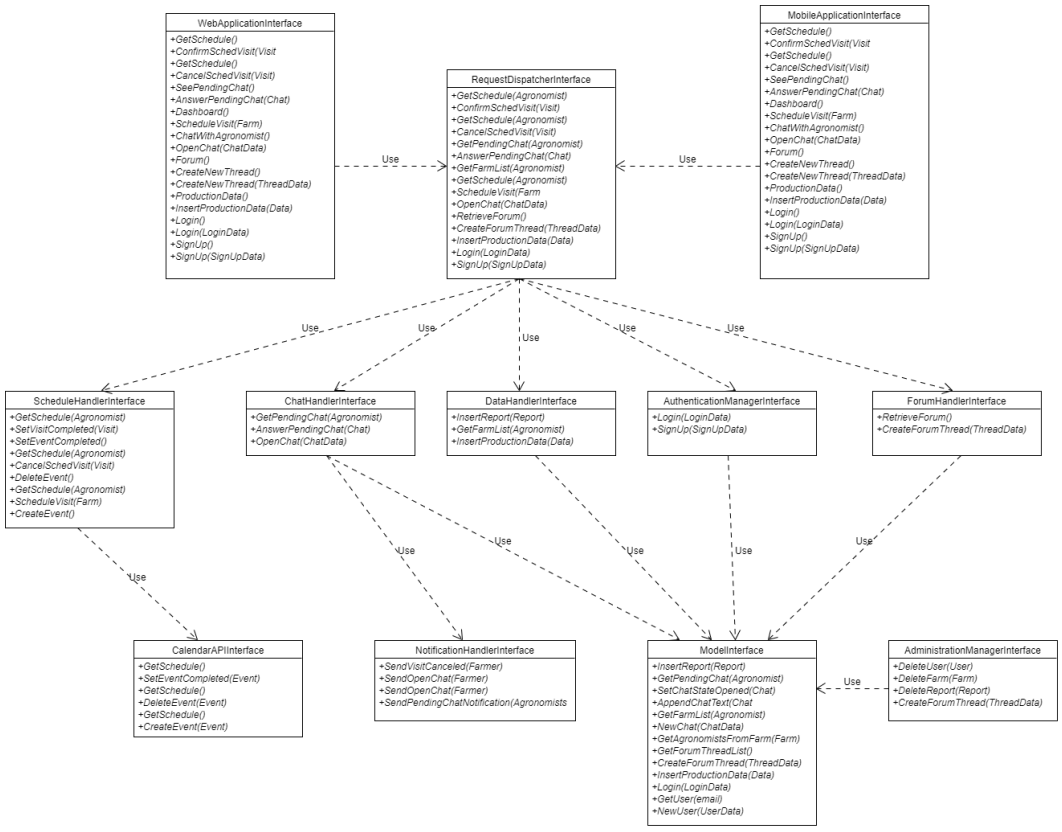

Figure 2.16: Component Interface

2.7 Selected Architectural Styles and Patterns

2.8 Other Design Decisions

PP Tier and Data Tier same cloud.

Chapter 3

User Interface Design

Chapter 4

Requirements Traceability

Chapter 5

Implementation, Integration, and Test Plan

Chapter 6

Effort Spent