Software Design Document (SDD)

2IPE0 SOFTWARE/WEB ENGINEERING PROJECT

November 8, 2019

GROUP 1

L. van de Beek	1015669
K.K.J. Burgers	1004884
M. Ghanem	1036435
A.N. Groote Woortmann	1000308
M.J.B. Keizer	0988425
B. van Leeuwen	0996101
L.A. Roozen	0948743
W.J.A. van Santvoort	1010775
H.P.A. Swinkels	1005900

TU/e

version: 3.2

supervisor: Serguei Roubtsov

customer: Dimas Satria

Abstract

This document is the Software Design Document (SDD) for CloudFarmer, a web application that delivers insights to farm related data developed by CloudFarmers as part of the Software Engineering Project at the Technical University of Eindhoven. This document complies with the ESA software standards [1]. Moreover, the design decisions provided fulfill the requirements in CloudFarmer's User Requirement Document. [3]

Contents

I Document Status Sheet

I.I General

Document title: Software Design Document (SDD)

Document identifier: SDD/3.2

Authors:

L. van de Beek	1015669
K.K.J. Burgers	1004884
M. Ghanem	1036435
A.N. Groote Woortmann	1000308
M.J.B. Keizer	0988425
B. van Leeuwen	0996101
L.A. Roozen	0948743
W.J.A. van Santvoort	1010775
H.P.A. Swinkels	1005900

Document status: Version 3.2

I.II Document History

Version	Date	Authors	Reason
0.1	17-09-2019	Willem	Initial version.
0.2	23-09-2019	Arthur	Added sections 1.1 & 1.2.
0.3	24-09-2019	Arthur& Kas	Added sections 2.3 & 2.4.
0.4	25-09-2019	Arthur& Kas	Added sections 2.1 & 2.2.
0.5	26-09-2019	Kas	Added sections 1.5 partly.
0.6	27-09-2019	Kas	Partly implemented section 3.4.
0.7	30-09-2019	Mohamed	Partly completed sections 3.6 & 4.
0.8	05-10-2019	Arthur & Kas & Mohamed	Completed 3.3.
0.9	06-10-2019	Luuk	Changed section 1.5, 3.1 and 3.2.
1.0	06-10-2019	Arthur	Completed first draft version.
1.1	15-10-2019	Luuk	Changed the UML class diagram in section 3.2.1

1.2	16-10-2019	Luuk	section 3.2.1
1.3	17-10-2019	Luuk	section 3.2.1
1.4	20-10-2019	Bram	Rewritten section 3.6
2.0	17-10-2019	Willem	section 3.2.2
3.1	05-11-2019	Bram	Rewritten section 3.1.2, ER model description, improved 4.1, added 4.2
3.2	03-11-2019	Luuk	UML diagram update, rewritten section 3.4

II Document Change Records

Version	Date	Section	Reason
0.1	17-09-2019	All	Initial version.
1.0	06-10-2019	All	Draft version.
1.1	15-10-2019	Luuk	3.2.1
1.2	16-10-2019	Luuk	3.2.1
1.3	17-10-2019	Luuk	3.2.1
2.0	17-10-2019	Willem	3.2.2
3.2	03-11-2019	Bram & Luuk	3.1.2, 3.2, 3.4, 4.1, 4.2

1 Introduction

1.1 Purpose

The purpose of the Software Design Document (SDD) is to provide an overview of CloudFarmer in terms of function, purpose, environment, and relation to other systems as well as previous, current, and successor systems. Furthermore, this document aims to provide a high-level architectural overview of the application. It has been decomposed into separate components in order to describe each of its different aspects accordingly. For each of these components a technical description, interface dependencies, and interface relations of both the other components and the external are provided. The design decisions taken are described and motivated, as well as an estimate for the required resources to run the application and the web server. This document is primarily intended for professionals in the software development sector.

1.2 Scope

CloudFarmers is a team of Computer Science Bachelor students working on a Software Engineering Project for the TU/e and Dimas Satria. Dimas Satria is a representative of *Praktijkcentrum Voor Precisie Landbouw*, a centre founded to speed up the adoption of precision agriculture (PA) in the Netherlands.

Precision agriculture is a farming management concept based on observing, measuring and responding to inter and intra-field variability in crops. The goal of precision agriculture research is to define a decision support system (DSS) for the whole farm management with the goal of optimising returns on inputs while preserving resources.

The goal of the CloudFarmers is to develop a web application that gives users of the application insights into the information of specific fields. These users might be farmers, researchers or even just curious people. This information will be accessible through an easy to use user interface that displays data gathered from a number of different data sources in an orderly manner.

1.3 List of definitions

1.3.1 Definitions

Name	Definition

Active farm	An active farm is the farm to which the map, history, live, field, edit data and farm settings view are currently referencing.
Active field	An active field is the field to which the field view is currently referencing.
Active crop field	An active crop field is the crop field to which the field view is currently referencing.
Available farm	A farm of which the user is allowed to see the information.
CloudFarm	Database server for the application.
CloudFarmer	Application as defined by the project group and the customer.
CloudFarmers	The project group tasked to create the aforementioned application.
Crop	The virtual representation of a physical crop.
Crop field	The virtual representation of a physical crop field. A field can contain multiple crop fields.
Crop field information/ the information of a crop field	All the information about a specific crop field; location, crop type, crop year, crop season, area.
Customer	The person who assigned the task to create the application to the project group. In this case Dimas Satria.
Dacom	The company that provided us with their crop database and their API for extracting information from this database.
Data sources	List of all the sources of data that are known by the application.
Equipment	The sensors in a field.
Farm	The virtual representation of a physical farm. A user is allowed to have more than one virtual farm.
Farmer (Role)	A person who works on the physical farm.
Farm admin (Role)	A user with unrestricted access to all data associated with a specific farm. (E.g farm information/permissions, field information/permissions ... etc)

Farm information/ the information of a farm	All the information about a specific farm; ID number, name, address, postal code, country, email, phone number, website.
Farm permissions	A set of rules that determine which users are able to read, write or delete farm information.
Field	The virtual representation of a physical field. A farm can have multiple virtual fields.
Field information/ the information of a field	All the information about a specific field; location, field name, area, size in hectares, soil type.
Field permissions	A set of rules that determine which users are able to read, write or delete field information.
General user (Role)	A user that is a member of a farm without permissions.
Member	A user is a member of a farm, if they have a role assigned to them on that farm.
Observation	Metadata.
Observation Data	A general class of data referring to Dacom's data and/or WolkyTolky data and/or data input by the user.
Physical crop field	A physical crop field is a physical field that contains at least one type of crop.
Physical farm	An area of land that is devoted to agricultural processes, it is the basic facility for food.
Physical field	A physical field is a patch of dirt, clay, or some type of rock that is owned by a farmer, on which they are able to grow crops.
Props	A collection of data variables. Most of the time use abbreviate which variables are sent to a function.
Researcher (Role)	A person who carries out academic or scientific research.
Selected farm	A farm that the user has selected.
Slug	A set of attributes of an equipment model : brand name, model, model year, series, version.
User	A person that interacts with the application.
User role	A role that can be assigned to a user within a farm which defines all the field permissions and farm permissions of the user. These different roles are defined in the definitions and have the tag (role) at the end of them.

Views	The map view, history view, live view, fields view, settings view and personal settings view.
Widgets	A component of an interface that enables a user to perform a function or access a service.

1.3.2 Abbreviations and Acronyms

Acronym	Meaning
CSV	Comma-Separated Values (file format)
DIPPA	Data Integration Platform for Precision Agriculture
ESA	European Space Agency
DOM	Document Object Model
ID	Idobject
JSON	JavaScript Object Notation
JWT	JSON Web Tokens
OAuth	Open authorization
PDEng	Professional Doctorate Engineering
PNG	Portable Network Graphics (file format)
SEP	Software Engineering Project
TU/e	Eindhoven University of Technology
URD	User Requirements Document
XML	Extensible Markup Language

1.4 List of References

[1] *E.B. for software Standardisation and Control*. ESA software standards, 1991.
[2] B. Enkhtaivan. "DIPPA : Data integration platform for precision agriculture". English. PDEng thesis. PhD thesis. Oct. 2018.
[3] *User Requirements Document*. Technische Universiteit Eindhoven, 2019.
[4] *Agrosmart Homepage*. URL: `https://agrosmart.com.br/en/`.
[5] *AngularJS*. URL: `https://angular.io`.
[6] *Dacom Homepage*. URL: `https://www.dacom.nl`.
[7] *Dart*. URL: `https://dart.dev`.

[8] *Express*. URL: https://expressjs.com.
[9] *Farmersedge Homepage*. URL: https://www.farmersedge.ca/.
[10] *Flow*. URL: https://flow.microsoft.com/en-us/.
[11] *Gitlab*. URL: https://about.gitlab.com.
[12] *How Fast Should A Website Load in 2019?* URL: https://www.hobo-web.co.uk/your-website-design-should-load-in-4-seconds/.
[13] *Jest*. URL: https://jestjs.io.
[14] *Kotlin*. URL: https://kotlinlang.org.
[15] *Laravel*. URL: https://laravel.com.
[16] *Material UI*. URL: https://material-ui.com.
[17] *Measure Performance with the RAIL Model*. URL: https://developers.google.com/web/fundamentals/performance/rail.
[18] *Mock data explanation*. URL: https://en.wikipedia.org/wiki/Mock_object.
[19] *NodeJS*. URL: https://nodejs.org/en/.
[20] *PHP*. URL: https://www.php.net.
[21] *PostgreSQL*. URL: https://www.postgresql.org.
[22] *Python*. URL: https://www.python.org.
[23] *React*. URL: https://reactjs.org.
[24] *Ruby On Rails*. URL: https://rubyonrails.org.
[25] *Typescript*. URL: http://www.typescriptlang.org.
[26] *User-centric Performance Metrics*. URL: https://developers.google.com/web/fundamentals/performance/user-centric-performance-metrics.
[27] *VueJS*. URL: https://vuejs.org.
[28] *WebAssembly*. URL: https://webassembly.org.

1.5 Overview

The remainder of this document consists of three chapters. Chapter two provides a general description of CloudFarmer that outlines its relation to other systems as well as its relation to previous, current and future projects. Chapter two also describes the purpose and function of the application and its operational environment.

Chapter three provides a description of the system architecture of CloudFarmer. In section 3.1 an overview of the different modules present in the application is given. In section 3.2 a UML class diagram of the application is presented, as well as the explanation of the purpose of each function and variable present in this diagram. Section 3.3 presents the dynamics inside the application that are described based on the UML sequence diagrams. In section 3.4 the different data types that are implemented into the application are explained. Section 3.5 discusses the communication between the application and outside sources, and lastly in section 3.6 the design decisions made during the development of this application are explained.

Chapter four provides the reader with an overview of the requirements that are needed to run the application, as well as the performance of the application.

2 System overview

2.1 Relation to current projects

CloudFarmer is an online web application that shows data gathered from farms. CloudFarmer is closely related to the DIPPA project[2], DIPPA is a precision agriculture application developed at the TU/e and is a free to use open-source software system that is focused on farm management. CloudFarmer differs from DIPPA is that it will be more focused on being a information discovery system instead of being a farm management system. Consequently, it is going to differ from DIPPA in terms of implementation. Cloudfarmer will also focus on making the data accessible for all kind of users, even general users and researchers. To create an user friendly interface for all these users, CloudFarmer will be designed so users can easily switch between farms and have clear sections on where what data is.

CloudFarmer can also be used as a farm management system. Therefore, we see other companies who have this functionality as competitors. Some of the biggest competitors are Agrosmart[4], Dacom[6], and Farmersedge[9]. The difference between CloudFarmer and the software of these companies is that CloudFarmer will be a free to use open source application and CloudFarmer will not give you advice related to crop management but rather it will show data. Since the application is free to use we expect that there will be a bigger audience. Besides, CloudFarmer's users are not restricted to people in the farming sector.

For a lot of farmers the use of the already existing software it too complicated or too expensive to be helpful to get insights into their farm data. CloudFarmer provides a way to circumvent this as it allows these farmers to observe other farms and learn from their data. Furthermore, it is possible to use our software without connecting the sensors because it is possible to upload the data manually to the server.

2.2 Relation to predecessor and successor projects

The predecessor of CloudFarmer is DIPPA. DIPPA was created as a final design project for the Professional Doctorate in Engineering program at the TU/e. DIPPA is also a precision agriculture application which collects, saves and shows data from its data sources.
There are currently no concrete plans for successor projects. Since CloudFarmer will be open source software the likelihood of people changing the code and implementing new features is very high. These changes can include features like widgets and having multiple active farms to compare data between farms more easily.

2.3 Function and purpose

In recent years, sensors have become a highly adopted utility in the world of agriculture. Sensors can measure a variety of variables, like humidity, crop yield or terrain features

of a farm field. To get a clear picture of what is happening on a farm a piece of software can be developed that will collect the data from the sensors, store it in a database, and display this to an user. This way an user gets a better understanding of the different variables involved in the equality and the quantity of crops. The problem is that such software is usually not free.

CloudFarmer, nonetheless, will provide an entirely free alternative for these kind of software programs. While incorporating a user friendly interface. Our site can be used by farmers,researchers all over the world, but also by people who are just interested in this subject. CloudFarmer is unique because the site is based data gathering and will not give a biased advise on what to do. To achieve this, CloudFarmer will incorporate the following functions:

- Constructing a data-overview: CloudFarmer combines the data from Dacom and our own database, CloudFarm, to make an understandable data-overview which the user can use to analyse the data with ease.
- Showing a Data-overview: CloudFarmer provides an endpoint so that the data of the fields can be retrieved and analysed with ease.
- Importing data: CloudFarmer allows the user to upload the data in the form of a CSV file which can be obtained from the sensors directly.
- CloudFarmer allows the user to download the data-overview in the form of a CSV file.

2.4 Environment

CloudFarmer will be used in a web-based environment. It will run on Chrome 72 and later versions and will be stored on the server. The following types of users can be distinguished; people who own farms, people who work on farms, people who want to research farms or simply just people who have an interest in farms and their data. We describe these four parties in the following terms respectively:

Farm admin
: This is the person who created the farm. The farm admin has the most rights on a farm compared to other users. They are the only ones that can read, edit and change other users' roles on that farm. Farm admins are also the only users that are allowed to edit and delete farm data. Furthermore, they have the rights to read, edit, and delete data from the field, crop field, observation data, and equipment data.

Farmer
: A farmer will most likely work on the physical farm. A farmer has fewer rights than a farm admin but more rights than a general user and a researcher on a farm. A farmer cannot edit nor delete farm data and they cannot change the roles users have. However, they are allowed to read farm data and they have the rights to

read, edit, and delete values pertaining to field data, crop field data, observation data and equipment data.

Researcher
A researcher is concerned with all the information regarding a farm. Thus, a researcher is allowed to read farm data, field data, crop field data, observation data, and equipment data. However, a researcher can have more rights on a farm if the farm admin allows this; the researcher might be eligible to edit and/or delete observation data of a farm.

General user
A general user is not related to the farm and is merely curious about the farm and its data. A general user has the least rights; they can only see the farm name and the country where the farm is located. Moreover, if a farm admin has set the farm data to public then a general user can view the farm data, field data, crop field data, observation data, and equipment data.

Finally, a user can have multiple different roles on multiple different farms, e.g. a user can be a farm admin on one farm and a farmer on another. The role a user has on a farm is decided by the farms farm admin. A user is categorised as a general user if they were not assigned a role in a farm by the farm admin.

2.5 Relation to other systems

CloudFarmer will be a standalone application, however it uses a few third party services to collect data. These are the Dacom API, WolkyTolky API, and the API provided by the customer via Swaggerhub.

2.5.1 Dacom API

CloudFarmer utilizes that API to extract data from Dacom's database. This data is a foundation of CloudFarmer's database since it incorporates data from previous time periods. Nevertheless, Dacom's API will be discarded when CloudFarmer's database is large enough as data from previous time periods will not be of high importance.

2.5.2 WolkyTolky API

The WolkyTolky API is for getting the live data from the data sensors. By using the WolkyTolky API farmers don't need to add their data manually, which will save time. The farm admin or farmer needs to add a datamap the first time it uses the WolkyTolky API to get data from its sensors. After this the datamap will be saved and the data from the WolkyTolky API regarding the aforementioned sensor will be saved automatically.

User Auth API

The User Auth module checks which rights the user has and if the user has the permission to use another service module. It also keeps track and manages the user data. As all service modules need permission to access their corresponding database, they are all linked to the User Auth module. The User Auth module also connects to a singular database that keeps a record of all user information stored by the application and the associated roles and permissions. A primary function of the User Auth module is to check the log-in credentials and distribute the correct requests to the other service modules to retrieve the user's dashboard data. The individual connections of the User Auth module to the other service modules are displayed in figure 4, shown in green.

Data Sensing API

The Data Sensing module handles most of the data requests from the front-end. It has a direct link to the Farm Data Storage to create, read, edit, and delete data, such as the farms, fields, cropfield, countries, equipments, and observations. It also handles the importing of new data by using data maps created by the Data Mapping module. To display a timeline of a specific data set, the Data Sensing module retrieves data from the Farm Data Storage and sends it to the Data Linking module. If a user imports their own data by uploading a CSV file, the Data Sensing module requests a confirmation from the Data Mapping module to see if the CSV file has correct data columns and if a valid data map exists.

Data Mapping API

The Data Mapping module controls all aspects of the data maps, i.e. creating a new data map and retrieving all existing data maps from the Farm Data Storage. If the user wants to upload new data to the Farm Data Storage, they need to select an existing data map or create a new one. The Data Mapping module handles the creation of this new data map by checking the validity of this data map. It can also retrieve a list of existing data maps from the Farm Data Storage to allow the Data Sensing module to create a selection for the user.

Data Linking API

The Data Linking module periodically retrieves live sensor data to create an up-to-date view of the farm information of a specific farm. The Data Linking module can connect to all kinds of different sensor APIs, if it has a correct data map from the Data Mapping module. Currently, the WolkyTolky API module and the DACOM API module are connected to the Data Linking module, but this number can increase due to its modularity. If there is a new data mapping path from the WolkyTolky API module, it automatically updates the corresponding data map in the Data Mapping module.

3 System architecture

3.1 Architectural design

The design of CloudFarmer is partitioned into front-end and back-end. The front-end part of the application will provide the user with the user interface. The back-end handles the application's logic as well as providing all the data to the front-end.

3.1.1 **Front-end**

The front-end consists of the top-bar and the side menu with the different views that can be loaded into the leftover space. The views displayed to the user depend on their roles and rights associated with them. For example, only a Farm admin can access the farm settings view.

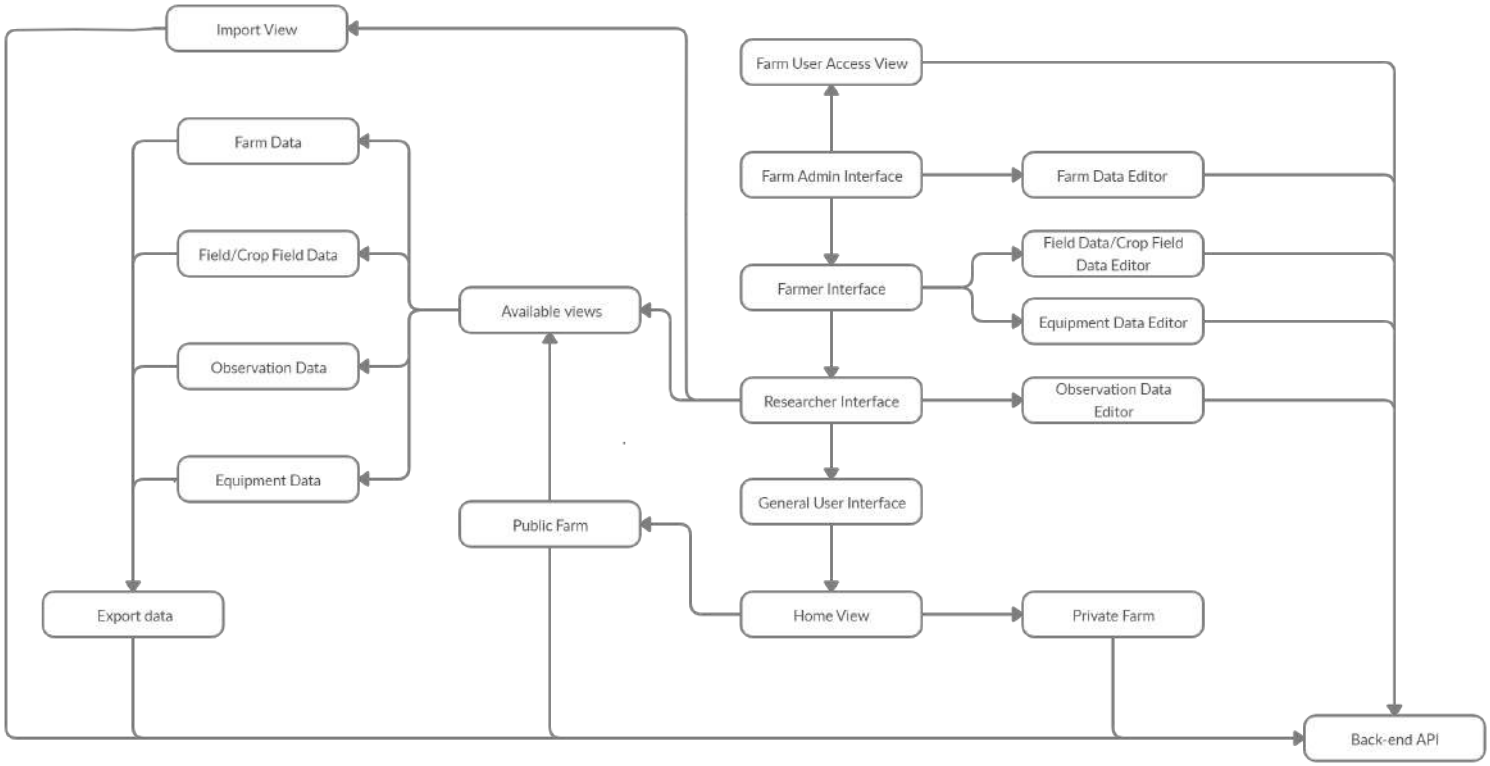

Figure 1: Front-end Modules

General User Interface

The General User Interface provides the general user with its visualisation and functionalities by using the Back-end API. It uses the Home View module to visualise the home page, that will be seen first upon logging in to the application. It will also use the Public Farm module to find all the public farms and give the general users viewing access to these farms.

Researcher Interface

The Researcher Interface extends on the functionalities of the General User Interface. It relies on Observation Data Edit, which communicates with the Back-end API, and the Available Views modules to visualise the editing of the observation, equipment, (crop)field, and farm data of the farm.

Farmer Interface

The Farmer Interface extends the Researcher Interface the same way as the Researcher Interface extends the General User Interface. It relies on (Crop)Field Data Edit and Equipment Data Edit modules to visualise the editing of field data, crop field data, and equipment data.

Farm Admin Interface

The Farm Admin Interface will provide the farm admin with all of the visualisation and the functionalities of the other interfaces combined. It uses the Farmer Interface, which in turn uses the Researcher Interface, which uses the General User Interface, which uses the back-end API. As the rights of researchers extend the rights of general user, the rights of farmers extend the rights of researchers, and the farm admin rights extend the rights of the farmers, we can model it as a decorator of each child interface. On top of that, the Farm Admin Interface also retrieves information from the Farm User Access View. The Farm Admin Interface is the only interface which implements this view, since it is the only role with those specific rights. The Farm Admin Interface relies on the Farm Data Edit module to visualise the editing of the farm data, which houses all general information of the farm, such as the email and website.

Farm User Access View

The Farm User Access View module encompasses the rendering and editing of user roles on that farm. This is done by communicating with the Back-end API and getting commands from the Farm Admin Interface. As the farm admin is the only role with the rights to see this view, the editing of the user roles is done here as well.

Farm Data Editor

The Farm Data Edit module allows farm admins to change the farm information data. This is done by using the Back-end API module which will get the farm data and will get the requests to change the data of a farm. The Farm Admin Interface can give commands to the Farm Data Edit module to change, edit, or delete farm data.

Field Data/Crop Field Data Editor

The Field Data/Crop Field Data Edit module allows farmers and farm admins to edit the shape of the fields and crop fields. This is done by using the Back-end API module, this module will provide the Field Data/Crop Field Data Edit module with the option to send change request to the back-end. These request are done by farmers or farm admin since these are the only groups of users that can access this module.

Equipment Data Editor

The Equipment Data Edit module allows farmers and farm admins to edit the equipment data. This is done by using the Back-end API module, this module will provide the Equipment Data Edit module with the option to send change request to the back-end. These requests are done by farmers or farm admin since these are the only groups of users that can access this module.

Observation Data Editor

The Observation Data Edit module allows researchers, farmers, and farm admins to edit the observation data. This is done by using the Back-end API module, this module will provide the Observation Data Edit module with the option to send change requests to the back-end through the API. These request are done by farmers, farm admin, or researchers since these are the only groups of users that can access this module.

Home View

The Home View module will visualise the first page a user enters once they have logged in, this page is a list of all the farms a user can see. This list is a list of all farms since users can always see the farm name and the country the farm is located. To see more information, a farm needs to have its data set to public or a user should at least be a researcher. To visualise this module the home view module will get information from the Public Farm module and from the Private Farm module. This way the Home View module has all the public and private farms and can visualise a table with all farms and showing only the data of the farm that is allowed.

Private Farm

The Private Farm module will contain a table of all the farms that have set their information to private. It will do this by getting the information from the Back-end API module.

Public Farm

The Public Farm module will contain a table of all the farms that have set their information to public. This means that general users can view their data from the farm, fields, crop fields, equipment, and observation. Together with the private farm list this will create a list of all the farms on out application.

Available Views

The Available Views module will combine all the users that are allowed to view the data of the selected farm. It gets its information from Researcher Interface and Public Farm module. Because a Farmer and a Farm Admin have extended rights from researcher we can say that all the views a researcher is allowed to see a farmer and a farm admin are also allowed to see.

Farm Data

The Farm Data module will visualise the data from the farm. This is done by getting information from the Back-end API module.

Field/Crop Field Data

The Field/Crop Field Data module will visualise the data from the fields and the crop fields inside of the fields. This is done by getting the data from the Back-end API module.

Observation Data

The Observation Data module will visualise the data from the observations done by the sensors. This is done by getting the data from the Back-end API module.

Equipment Data

The Equipment Data module will visualise the data from the sensors. This is done by getting the data from the Back-end API module.

Back-end API

The back-end API module will provide the communication with the back-end and the service modules, so that the front-end has the correct data to display.

Import View

The Import View module visualises the view where farm admins, farmers, and researchers can import data. This will then give this data to the Back-end API to save it.

Export Data

The Export Data module will visualise the option for users to export the data from the different views. This is done with communication with the Back-end API to get the data.

3.1.2 Back-end

The back-end of the application consist of multiple APIs, services, and databases. The back-end module is a smaller subcomponent that handles the connection between the front-end and the services. The services, or service modules, comprise of the Data Linking API module, Data Sensing API module, Data Mapping API module, and User Auth API module. For future reference, the back-end encompasses the entire model (excluding the front-end module) and the back-end module handles the communication between the service modules. Figure 2 shows all relations that the front-end module, back-end module, service modules, Cloud API modules, and databases have.

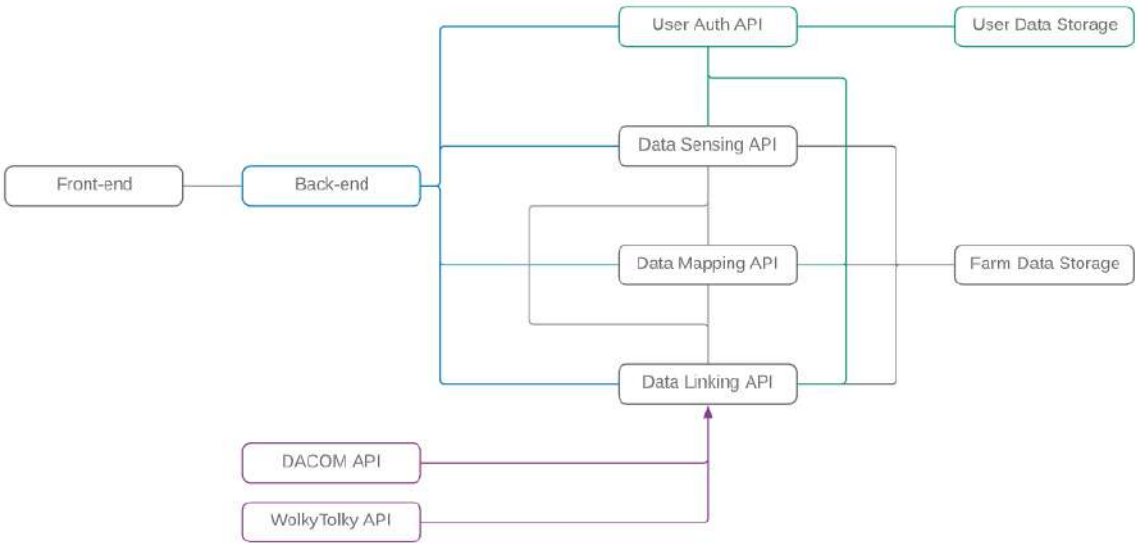

Figure 2: Back-end modules

Front-end

The front-end module handles all of the interactivity between the user and the application. The front-end module sends requests for data or data handling to the back-end module. These requests will also be received again from the back-end module. The other components of the front-end module are described in section 3.1.1.

Back-end

The back-end module is essentially the bridge between the front-end and the service modules. In figure 3 it shows that the back-end module communicates with four different service modules. These include the Data Linking API module, Data Sensing API module, Data Mapping API module, and User Auth API module. These service modules also interact with their corresponding database and with each other, shown in figure 4. The User Auth module (shown in green) connects to all other service modules directly, as these modules need to check the correct user permissions before the requests are being handled and arranged in the database.

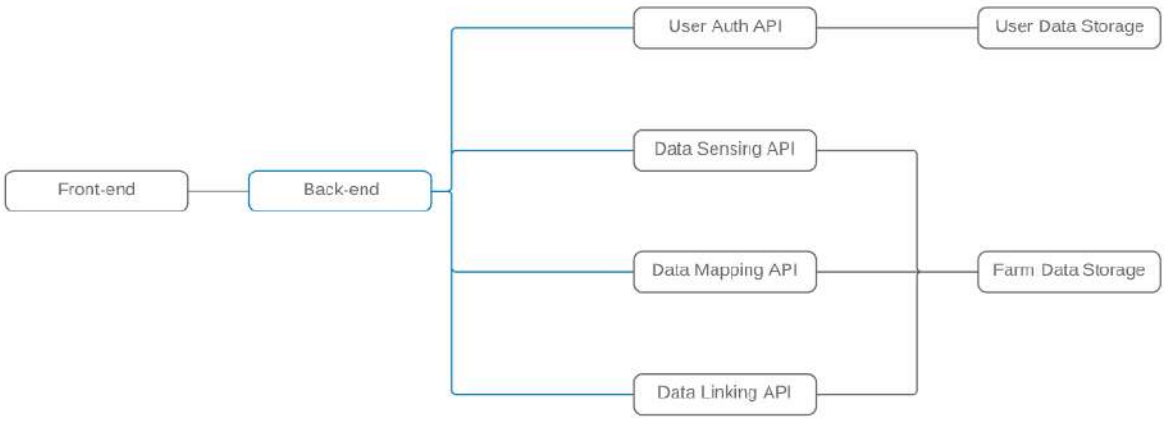

Figure 3: Base back-end modules

User Auth

The User Auth module checks which rights the user has and if the user has the permission to use another service module. It also keeps track and manages the user data. As all service modules need permission to access their corresponding database, they are

all linked to the User Auth module. The User Auth module also connects to a singular database that keeps a record of all user information stored by the application and the associated roles and permissions. A primary function of the User Auth module is to check the log-in credentials and distribute the correct requests to the other service modules to retrieve the user's dashboard data. The individual connections of the User Auth module to the other service modules are displayed in figure 4, shown in green.

Data Sensing

The Data Sensing module handles most of the data requests from the front-end. It has a direct link to the Farm Data Storage to create, read, edit, and delete data, such as the farms, fields, cropfield, countries, equipments, and observations. It also handles the importing of new data by using data maps created by the Data Mapping module. To display a timeline of a specific data set, the Data Sensing module retrieves data from the Farm Data Storage and sends it to the Data Linking module. If a user imports their own data by uploading a CSV file, the Data Sensing module requests a confirmation from the Data Mapping module to see if the CSV file has correct data columns and if a valid data map exists.

Data Mapping

The Data Mapping module controls all aspects of the data maps, i.e. creating a new data map and retrieving all existing data maps from the Farm Data Storage. If the user wants to upload new data to the Farm Data Storage, they need to select an existing data map or create a new one. The Data Mapping module handles the creation of this new data map by checking the validity of this data map. It can also retrieve a list of existing data maps from the Farm Data Storage to allow the Data Sensing module to create a selection for the user.

Data Linking

The Data Linking module periodically retrieves live sensor data to create an up-to-date view of the farm information of a specific farm. The Data Linking module can connect to all kinds of different sensor APIs, if it has a correct data map from the Data Mapping module. Currently, the WolkyTolky API module and the Dacom API module are connected to the Data Linking module, but this number can increase due to its modularity. If there is a new data mapping path from the WolkyTolky API module, it automatically updates the corresponding data map in the Data Mapping module.

Farm Data Storage

The Farm Data Storage module is the main database to store all data regarding the farms, fields, cropfields, countries, equipments, data maps, and observations. It is con-

nected to three of the four service modules; the Data Sensing module, Data Mapping module, and the Data Linking module, seen in figure 4. This allows the service modules to independently retrieve their specific data sets from the Farm Data Storage module. It is, however, not connected to the User Auth module, as the personal user information is required to be separate from the rest of the farm data, because of privacy.

User Data Storage

The User Data Storage module is the database that is used for all data regarding users. This also includes what roles and/or rights they have on a farm. It is only linked to the User Auth module, as seen in figure 4. This way, we can provide all the farm related data to the other service modules and keep the user information and permissions separate. By splitting up the databases, we can ensure the user information database is referenced as little as possible and allow for a more secure API to handle this information.

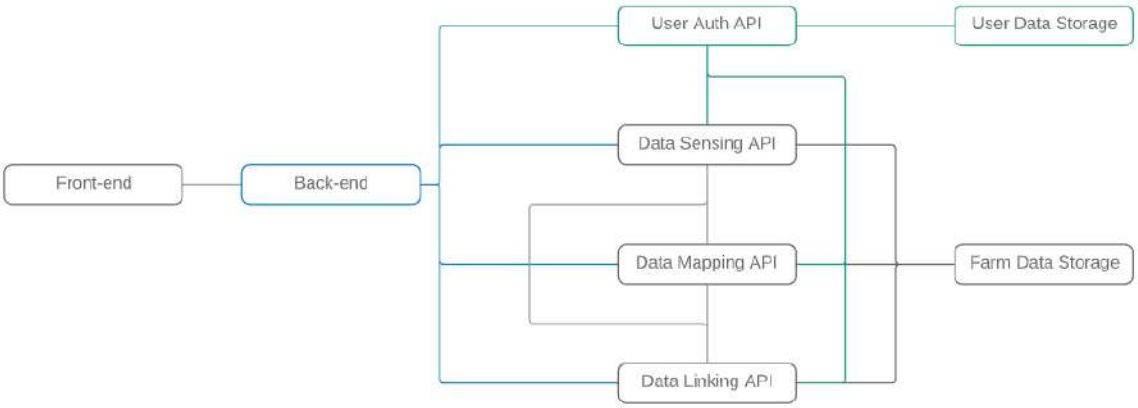

Figure 4: Connected back-end modules

Dacom API

The Dacom API module will retrieve data from the Dacom database. It is currently connected to the Data Linking module to combine this data with the incoming data of the live APIs and ensure that the data is correctly mapped, as seen in figure 5. The Dacom API module will eventually be phased out, as then the application will solely rely on the newly generated live data from the live APIs.

WolkyTolky API

The WolkyTolky API module will give live observation data for each farm field that the WolkyTolky API has data from. It is linked only to the Data Linking module, as seen in

figure 5. This way, the module's incoming data will be correctly merged with the already existing data in the Farm Data Storage module from the Dacom API, the WolkyTolky API itself, and other live APIs.

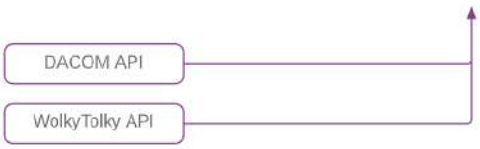

Figure 5: Cloud API modules

3.2 Logical model description

3.2.1 Front-end

This section contains an explanation of the front-end by means of UML diagrams. In the first paragraph the model described in figure 6 is explained. The controllers are the part of the application that make calls to the API provided by the client, as well as to the back-end which can be found in the next section. This figure also contains the object classes that are needed by these controllers. In the second paragraph the model described in figure 7 is explained. These are the classes that are used to create the user interface, in this diagram references are made to the classes described in figure 6. This is done, to explain which part of the user interface needs which controller(s).

3.2.1.1 Controllers

In this section an explanation is given for each function and variable that can be found in figure 6. The **«Database TU/e»** object refers to the database as provided by the client. Whereas the **«MicroServices»** object refers to the part described by the UML diagram in section 3.2.2.

«Controller»: this class describes the abstract controller class, that is the variables and functions that each instance of this class needs.

parameters

- *apiURLTUeServer: string* ⇒ This variable contains the URL to the API that is needed to communicate with the database of the client.
- *JWT: string* ⇒ This variable contains the JSON web token that is needed to verify that the user making the request has the correct permissions.

- *refreshToken: string* ⇒ This variable contains the token that is used to refresh the JWT once it expires.

functions

- *handleErrors(code:integer): void* ⇒ This function handles common response errors, such as a 404 or a 501.

Role controller

functions

- *getAllUsersWithRoleOnFarm(activeFarmId: integer): Array<integer, string>* ⇒ This function returns all the users that have a role on the active farm.
- *addUserWithRoleToFarm(activeFarmId: integer, email: string, role: string): void* ⇒ This function adds a user with a role to the active farm.
- *getUserRoleOnFarm(activeFarmId: integer, userId: integer): string* ⇒ This function gets the role of a specific user on the active farm.
- *changeUserRoleOnFarm(activeFarmId: integer, userId: integer, role: string): void* ⇒ This function changes the role of a user on the active farm.
- *getUserRole(): Array<string>* ⇒ This function returns all the possible roles.

User controller

parameters

- *userId: integer* ⇒ This variable stores the id of the user, to which the user controller is referencing.

functions

- *login: email: string, password: string): 0 | 1 | 2* ⇒ This function verifies that the credentials are correct, and returns a code based on the response. If the credentials are correct, then the JWT is set as well as the refreshToken.
- *logout(userId: integer): void* ⇒ This function removes the JWT and the refreshToken.
- *register(Props): 0 | 1 | 2* ⇒ This function registers a user with the database, and returns a code based on the response it receives.
- *loggedIn(): boolean* ⇒ This function checks whether a user is already logged in.
- *refreshToken(refreshToken: string): Array<string>* ⇒ This function refreshes the JWT.
- *getUser(JWT: string, userId: integer): User* ⇒ This function gets the user object belonging to the userId variable.
- *updateUser(JWT: string, user: User): boolean* ⇒ This function updates the user information.

- *deleteUser(JWT: string, userId: integer): boolean* ⇒ This function deletes a user.

User

parameters

- *userId: string* ⇒ This variable contains the unique the id of this user object.
- *email: string* ⇒ This variable contains the email belonging to this user object.
- *firstName: string* ⇒ This variable contains the first name belonging to this user object.
- *lastName: string* ⇒ This variable contains the last name belonging to this user object.
- *createDate: string* ⇒ This variable contains the creation date of this user object.

functions

- *onCreate(Props): void* ⇒ when this object is instantiated this function sets the variables.
- *getUserId(): integer* ⇒ This function returns the value of the userId variable.

DataMap controller

parameters

- *apiMicroService: string* ⇒ This variable contains the URL to the API created by us.

functions

- *getDataMaps(): Array<DataMap>* ⇒ This function returns all the available data maps.
- *getDataMap(dataMapId: integer): DataMap* ⇒ This function returns a specific data map.
- *addDataMap(dataMap: DataMap): void* ⇒ This function adds a new data map to the database.
- *updateDataMap(dataMap: DataMap): void* ⇒ This function updates a data map.
- *deleteDataMap(dataMapId: integer): void* ⇒ This function deletes a data map from the database.
- *import(dataMap: DataMap, csv: string): boolean* ⇒ This function can be used to import data into the database.

DataMap

parameters

- *dataMapId: integer* ⇒ This variable contains the unique id of this data map object.
- *name: string* ⇒ This variable contains the name of this data map object.
- *description: string* ⇒ This variable contains the description of this data map object.
- *hasHeader: boolean* ⇒ This variable contains whether the data has headers.
- *hasCoordinate: boolean* ⇒ This variable contains whether the data has coordinates.
- *hasData: boolean* ⇒ This variable contains whether the data has a date.
- *hasTime: boolean* ⇒ This variable contains whether the data has a time.
- *modelId: integer* ⇒ This variable contains the id of which model was used to collect the data.
- *accessibility: "private"* | *"public"* ⇒ This variable contains whether the data map is use able by everybody or only certain members of the farm.
- *map: Array<string, Observation>* ⇒ This variable contains the mapping from a column name to an observation.

functions

- *onCreate(Props): void* ⇒ when this object is instantiated this function sets the variables.
- *getDataMapId(): integer* ⇒ This function returns the value of the dataMapId variable.

Observation

parameters

- *type: "environment"* | *"harvest"* | *"crop"* ⇒ This variable contains the type of the observation.
- *context: string* ⇒ This variable contains the context of the observation.
- *parameter: string* ⇒ This variable contains the parameter of the observation.
- *unit: string* ⇒ This variable contains the unit of the observation.
- *conditionParameter: string* ⇒ This variable contains the parameter of the condition of the observation.
- *conditionValue: integer* ⇒ This variable contains the value of the condition of the observation.

- *conditionUnit: string* ⇒ This variable contains the unit of the condition of the observation.

functions

- *onCreate(Props): void* ⇒ when this object is instantiated this function sets the variables.

Types controller

functions

- *getCountries(): Array<Country>* ⇒ This function returns all known countries in the TU/e database.
- *getSoilTypes(): Array<SoilType>* ⇒ This function returns all known soil types in the TU/e database.
- *getCropTypes(): Array<CropType>* ⇒ This function returns all known crop types in the TU/e database.

Country

parameters

- *countryId: integer* ⇒ This variable contains the unique id of this country object.
- *countryName: string* ⇒ This variable contains the name of this country object.
- *countryCode(): string* ⇒ This variable contains the code of this country object.

functions

- *onCreate(Props): void* ⇒ when this object is instantiated this function sets the variables.
- *getCountryId(): integer* ⇒ This function returns the value of the countryId variable.
- *getCountryName(): string* ⇒ This function returns the value of the countryName variable.
- *getCountryCode(): string* ⇒ This function returns the value of the countryCode variable.

SoilType

parameters

- *soilTypeId: integer* ⇒ This variable contains the unique id of this soilType object.
- *soilTypeName: string* ⇒ This variable contains the name of this soilType object.

- *soilTypeCode(): string* ⇒ This variable contains the code of this soilType object.

functions

- *onCreate(Props): void* ⇒ when this object is instantiated this function sets the variables.
- *getSoilTypeId(): integer* ⇒ This function returns the value of the soilTypeId variable.
- *getSoilTypeName(): string* ⇒ This function returns the value of the soilTypeName variable.
- *getSoilTypeCode(): string* ⇒ This function returns the value of the soilTypeCode variable.

CropType

parameters

- *cropTypeId: integer* ⇒ This variable contains the unique id of this cropType object.
- *cropTypeName: string* ⇒ This variable contains the name of this cropType object.
- *cropTypeCode(): string* ⇒ This variable contains the code of this cropType object.

functions

- *onCreate(Props): void* ⇒ when this object is instantiated this function sets the variables.
- *getCropTypeId(): integer* ⇒ This function returns the value of the cropTypeId variable.
- *getCropTypeName(): string* ⇒ This function returns the value of the cropTypeName variable.
- *getCropTypeCode(): string* ⇒ This function returns the value of the cropTypeCode variable.

Farm controller

functions

- *getFarms(): Array<Farm>* ⇒ This function returns all the farm available.
- *addFarm(farm: Farm): void* ⇒ This function adds a new farm to the database.
- *getFarm(farmId: integer): Farm* ⇒ This function returns a specific farm.
- *updateFarm(farm: Farm): void* ⇒ This function updates a specific farm.
- *deleteFarm(farmId: integer): void* ⇒ This function deletes a specific farm.

Farm

parameters

- *farmId: integer* ⇒ This variable contains the unique id of this farm object.
- *name: string* ⇒ This variable contains the name of this farm object.
- *address: string* ⇒ This variable contains the address of this farm object.
- *postalCode: string* ⇒ This variable contains the postal code of this farm object.
- *country: Country* ⇒ This variable contains the country of this farm object.
- *email: string* ⇒ This variable contains the email of this farm object.
- *phone: string* ⇒ This variable contains the phone number of this farm object.
- *website: string* ⇒ This variable contains the website of this farm object.
- *accessibility: "private" | "public"* ⇒ This variable contains whether everybody can see this farm object or only users with a specific role.

functions

- *onCreate(Props): void* ⇒ when this object is instantiated this function sets the variables.
- *farmId: integer* ⇒ This function returns the value of the farmId variable.

Field controller

functions

- *getFields(farmId: integer): Array<Field>* ⇒ This function returns all the available fields of a specific farm.
- *getField(farmId: integer, fieldId: integer): Field* ⇒ This function returns a specific field from a specific farm.
- *addField(field: Field): void* ⇒ This function adds a field to a farm.
- *updateField(field: Field): void* ⇒ This function updates a field.
- *deleteField(farmId: integer, fieldId: integer): void* ⇒ This function removes a field from a farm.

Field

parameters

- *fieldId: integer* ⇒ This variable contains the unique id of this field object.
- *fieldName: string* ⇒ This variable contains the name of this field object.
- *location: Array<integer>* ⇒ This variable contains the location of this field object.

- *size: integer* ⇒ This variable contains the size of this field object.
- *soilType: SoilType* ⇒ This variable contains the soil type of this field object.
- *accessibility: "private" | "public"* ⇒ this variable contains whether the field can be seen by everybody or only by users with a specific role.
- *farmId: integer* ⇒ This variable contains the id of the farm that this field belongs to.

functions

- *onCreate(Props): void* ⇒ when this object is instantiated this function sets the variables.
- *getFieldId(): integer* ⇒ This function returns the value of the fieldId variable.

CropField controller

functions

- *getCropFields(farmId: integer, fieldId: integer): Array<CropField>* ⇒ This function returns all the available crop fields from a specific field.
- *getCropField(farmId: integer, fieldId: integer, cropFieldId: integer): CropField* ⇒ This function returns a specific crop field belonging to a specific field.
- *addCropField(cropField: CropField): void* ⇒ This function adds a crop field to a specific field.
- *updateCropField(cropField: CropField): void* ⇒ This function updates a specific crop field.
- *deleteCropField(farmId: integer, fieldId: integer, cropFieldId: integer): void* ⇒ This function deletes a specific crop field.

CropField

parameters

- *CropFieldId: integer* ⇒ This variable contains the unique id of this crop field object.
- *CropFieldName: string* ⇒ This variable contains the name of this crop field object.
- *periodStart: date* ⇒ This variable contains the start date of this crop field object.
- *periodEnd: date* ⇒ This variable contains the end date of this crop field object.
- *cropType: CropType* ⇒ This variable contains the crop type of this crop field object.

- *location: Array<integer>* ⇒ This variable contains the location of this crop field object.
- *accessibility: "private"* | *"public"* ⇒ This variable contains the crop field can be seen by every user, or by users who have to correct role.
- *fieldId: integer* ⇒ This variable contains the id of the field to which this crop field belongs.

functions

- *onCreate(Props): void* ⇒ when this object is instantiated this function sets the variables.
- *getCropFieldId(): integer* ⇒ This function returns the value of the cropFieldId variable.

Equipment controller

functions

- *getEquipments(farmId: integer): Array<Equipment>* ⇒ This function returns all the equipments of a specific farm.
- *addEquipment(equipment: Equipment): void* ⇒ This function adds new equipment to the TU/e database.
- *getEquipment(equipmentId: integer): Equipment* ⇒ This function returns a specific equipment.
- *updateEquipment(equipment: Equipment): void* ⇒ This function updates a specific equipment.
- *deleteEquipment(equipmentId: integer): void* ⇒ This function deletes a specific equipment.
- *getEquipmentModels(): Array<EquipmentModel>* ⇒ This function returns all available equipment models.
- *addEquipmentModel(equipmentModel: EquipmentModel): boolean* ⇒ This function adds a new equipment model to the TU/e database.
- *getEquipmentModel(equipmentModelId: integer): EquipmentModel* ⇒ This function returns a specific equipment model.

Equipment

parameters

- *equipmentId: integer* ⇒ This variable contains the unique id of the equipment object.
- *equipmentName: string* ⇒ This variable contains the name of the equipment object.
- *equipmentDescription: string* ⇒ This variable contains the description of the equipment object.

- *equipmentModelId: integer* ⇒ This variable contains the id of the equipment model belonging to the equipment object.
- *manufacturingDate: date* ⇒ This variable contains the manufacturing date of the equipment object.
- *serialNumber: string* ⇒ This variable contains the serial number of the equipment object.
- *accessibility: "private" | "public"* ⇒ This variable contains whether everybody can see the equipment object, or only users with the correct role.
- *apiKey: string* ⇒ This variable specifies the apiKey to look at the data that the equipment object is providing.
- *farmId: integer* ⇒ This variable contains the id of the farm to which the equipment object is linked.

functions

- *onCreate(Props): void* ⇒ when this object is instantiated this function sets the variables.
- *getEquipmentId(): integer* ⇒ This function returns the value of the equipmentId variable.

EquipmentModel

parameters

- *equipmentModelId: integer* ⇒ This variable contains the unique id of the equipment model object.
- *brandName: string* ⇒ This variable contains the name of the brand of the equipment model object.
- *model: string* ⇒ This variable contains the model of the equipment model object.
- *modelYear: string* ⇒ This variable contains the year of the equipment model object.
- *series: string* ⇒ This variable contains the series of the equipment model object.
- *softwareVersion: string* ⇒ This variable contains the software version of the equipment model object.
- *description: string* ⇒ This variable contains the description of the equipment model object.

functions

- *onCreate(Props): void* ⇒ when this object is instantiated this function sets the variables.

- *getEquipmentModelId(): integer* ⇒ This function returns the value of the equipmentModelId variable.

FarmData controller

parameters

- *apiMicroService: string* ⇒ This variable contains the URL to the API created by us.

functions

- *getLiveData(equipmentId: integer): string* ⇒ This function gets the live data using the apiKey from a specific equipment object.
- *saveLiveData(Props): void* ⇒ This function saves the live data it receives to the TU/e database.
- *getHistoryData(Props): Array<string, Observation>* ⇒ This function returns the data already stored in the TU/e database, and parses it using a data map.

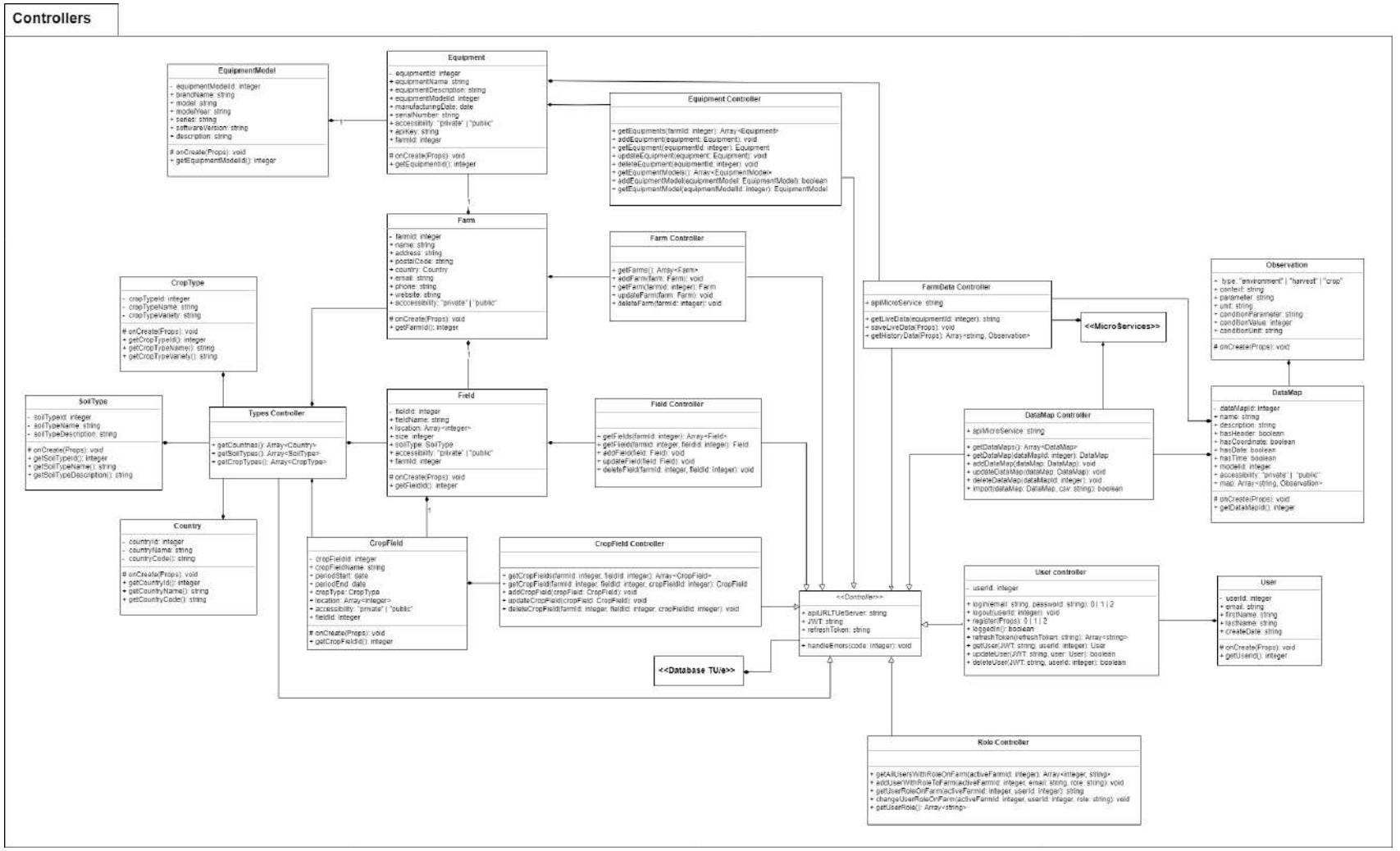

Figure 6: UML class diagram for the controllers.

3.2.1.2 User Interface

In this section an explanation is given for each function and variable that can be found in figure 7. Each class that has the tag *«Controllers»* above it, refers to the class specified in the previous section. For example the class **Field controller**, refers to the class with the same name in figure 6.

«View»

parameters

- *id: integer* ⇒ This variable contains the unique id of each view, so it can be referenced to.

functions

- *render(): void* ⇒ This function renders the UI components of the view into the browser.

General

parameters

- *activeFarmId: integer* ⇒ This variable contains the id of the farm that is currently active.
- *currentView: integer* ⇒ This variable contains the id of the view that is currently being displayed.
- *userRole: string* ⇒ This variable contains the role the user has on the active farm.
- *favorites: Array<integer>* ⇒ This variable contains the favorite farms of the user.

functions

- *render(): void* ⇒ This function renders the UI components of the view into the browser.
- *createFavoritesBar(favorites: Array<integer>): string* ⇒ This function creates the favorites bar UI component.
- *createSideMenu(userRole: string, activeFarmId: integer): string* ⇒ This function creates the side menu UI component.
- *renderViewInFragment(currentView: integer): void* ⇒ This function renders the current view into a fragment, so the user can user it.
- *isFavorite(farmId: integer): boolean* ⇒ This function returns whether the specified farm is a favorite farm.

PersonalSettingsView

parameters

- *user: User* ⇒ This variable contains the user object belonging to the currently logged in user.

functions

- *createUserInformationForm(): void* ⇒ This function creates a UI component on which the user can see their information.
- *createEditUserInformationForm(): void* ⇒ This function creates a UI component which the user can use to change their information.

Registration View

parameters

- *email: string* ⇒ This variable contains the email the user used to register themselves.
- *password: string* ⇒ This variable contains the password the user used to register themselves.
- *firstName: string* ⇒ This variable contains the first name the user used to register themselves.
- *lastName: string* ⇒ This variable contains the last name the user used to register themselves.

functions

- *render(): void* ⇒ This function renders the UI components of the view into the browser.
- *createRegistrationForm(): string* ⇒ This function creates the UI component that the user uses to register themselves.

user controller

parameters

- *email: string* ⇒ This variable contains the email the user used to log in.
- *password: string* ⇒ This variable contains the password the user used to log in.

functions

- *render(): void* ⇒ This function renders the UI components of the view into the browser.
- *createLoginForm(): string* ⇒ This function creates the UI component that the user can use to log in.

FarmSettings View

parameters

- *farm: Farm* ⇒ This variable contains the farm object belonging to the currently active farm.

- *usersAndRole: Array<integer, string>* ⇒ This variable contains all the userId's with their respective role on the active farm.

functions

- *createUserManagementTable(): string* ⇒ This function creates the UI component that the user can user to alter the roles of users on the active farm.
- *createUpdateFarmForm(): string* ⇒ This function creates the UI component the user can use to update their farm information.

Farm List View

parameters

- *farms: Array<Farm>* ⇒ This variable contains all the farm object the user is allowed to see.

functions

- *createFarmList(): string* ⇒ This function creates the UI component on which the user can see all the farms.
- *createUpdateFarmForm(): string* ⇒ This function creates the UI component the user can use to update their farm information.
- *addFavorite(farmId: integer): void* ⇒ This function adds a farm to a user's favorites.
- *deleteFavorite(farmId: integer): void* ⇒ This function removes a farm from the user's favorites.

Equipment View

parameters

- *equipments: Array<Equipment>* ⇒ This variable contains all the equipments a user is allowed to see.
- *equipmentModels: Array<EquipmentModel>* ⇒ This variable contains all the equipment models a user is allowed to see.

functions

- *createEquipmentList(): string* ⇒ This function creates the UI component which the user can use to view all the available equipments.
- *createUpdateEquipmentForm(): string* ⇒ This function creates the UI component which the user can use to edit the information of a specific equipment.
- *createEquipmentModelsList(): string* ⇒ This function creates the UI component which the user can use to view the available equipment models.

- *createUpdateEquipmentModelForm(): string* ⇒ This function creates the UI component which the user can use to edit the information of a specific equipment model.

DataMap View

parameters

- *dataMaps: Array<DataMap>* ⇒ This varaible contains all the data maps a user is allowed to see.

functions

- *createDataMapsList(): string* ⇒ This function creates the UI component the user can use to view all the available data maps.
- *createNewDataMapForm(): string* ⇒ This function creates the UI component the user can use to create new data maps.

Field View

parameters

- *fields: Array<Field>* ⇒ This variable contains the fields the user is allowed to see.
- *activeFieldId: integer* ⇒ This variable contains the id of the active field.
- *activeCropFieldId: integer* ⇒ This variable contains the id of the active crop field.
- *cropFields: Array<CropField>* ⇒ This variable contains the crop fields the user is allowed to see.
- *dataMaps: Array<DataMap>* ⇒ This variable contains the data maps the user is allowed to see.
- *dataMap: DataMap* ⇒ This variable contains the data map that is selected to import data with.
- *csv: string* ⇒ This variable contains the CSV file from which data should be imported.
- *googleMapsApiKey: string* ⇒ This variable contains the API key for the google maps API.

functions

- *createFieldsMap(): string* ⇒ This function uses the goole maps API key to create the UI component which shows all the available fields and crop fields.
- *createFieldsList(): string* ⇒ This function creates a UI component to display the available fields and crop fields in a list.
- *createUpdateFieldForm(): string* ⇒ This function creates a UI component the user can use to update the information of a field.

- *createUpdateCropFieldFrom(): string* ⇒ This function creates a UI component the user can use to update the information of a crop field.
- *createImportForm(): string* ⇒ This function creates the UI component the user can use to import data from a CSV file.

History View

parameters

- *fields: Array<Field>* ⇒ This variable contains all the fields the user can see.
- *selectedFields: Array<integer>* ⇒ This varaible contains all the fields the user has selected.
- *cropFields: Array<CropField>* ⇒ This variable contains all the crop fields a user is allowed to see.
- *selectedCropFields: Array<integer>* ⇒ This variable contains all the crop fields a user has selected.
- *availableCropTypes: Array<string>* ⇒ This variable contains all the crop types that are available based on the selected crop fields.
- *selectedCropTypes: Array<string>* ⇒ This variable contains all the crop types that are selected.
- *equipments: Array<Equipment>* ⇒ This variable contains all the equipments a user is allowed to see.
- *selectedEquipments: Array<integer>* ⇒ This variable contains all the equipments that are selected.
- *data: Array<string, observation>* ⇒ This variable contains the data data has been gathered based on the selected inputs.

functions

- *createDropDownsForSelection(): string* ⇒ This function creates the UI components the user can use to select the filters for their data.
- *createGraphs(): string* ⇒ This function creates the UI component that displays the data in the form of a graph.

Live View

parameters

- *fields: Array<Field>* ⇒ This variable contains the fields the user is allowed to see.
- *cropFields: Array<CropField>* ⇒ This variable contains the crop fields the user is allowed to see.

functions

– *createDataList(): string* $\Rightarrow$ This function creates the UI component that shows the live data based on each field and its crop fields.

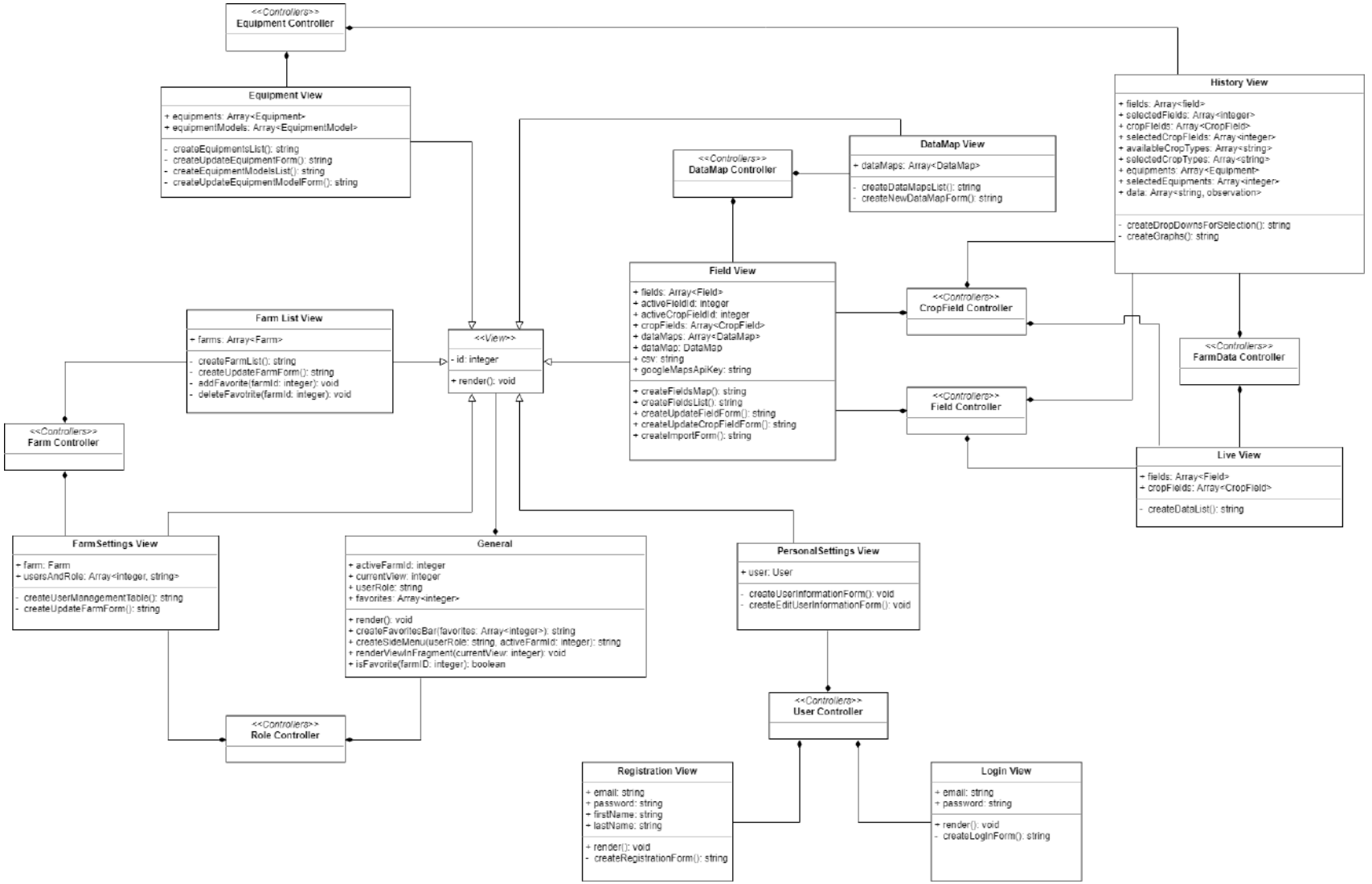

Figure 7: UML class diagram for the user interface.

3.2.2 Back-end

This section describes the Data Linking API using the UML diagram in figure 8. This section has no direct relations with the other sections, since the API runs separately from the UI application. The section contains the *Data Linking Controller* class which handles the incoming API requests and calls the functions needed from its child classes to return valid responses. These classes get their data via external API calls and the defined object classes.

Data Linking Controller This class handles incoming requests and calls the needed functions.

parameters:

- *req* ⇒ This is a parameter of type Request as defined in express. This type consists of all parameters send along with an API call.
- *res* ⇒ This is a parameter of type Response as defined in express. This type consists of all parameters send along with an API response.

functions:

- *GETequipmentsWt(): Response* ⇒ Calls the *getEquipmentsWt()* function from the *WolkyTolky Data Provider* class. It will return an object of Response type with in its body an array of *WolkyTolky Equipment* objects.
- *GETequipmentWt(req: Request): Response* ⇒ Calls the *getEquipmentWt(id: integer)* from the *WolkyTolky Data Provider* class with as the id req.query.id. It will return an object of Response type with in its body a *WolkyTolky Equipment* object.
- *POSTequipmentWt(req: Request): void* ⇒ Calls the *postEquipmentWt(obj: WolkyTolky Equipment)* function from the *WolkyTolky Data Provider* class with as obj a *WolkyTolky Equipment* object that is created based on the values in req.body. This function does not return a response.
- *PUTequipmentWt(req: Request): void* ⇒ Calls the *putEquipmentWt(obj: WolkyTolky Equipment)* function from the *WolkyTolky Data Provider* class with as obj a *WolkyTolky Equipment* object that is created based on the values in req.body. This function does not return a response.
- *DELETEequipmentWt(req: Request): void* ⇒ Calls the *deleteEquipment(id: integer)* function from the *WolkyTolky Data Provider* class with as id req.query.id. This function does not return a response.
- *GETequipmentIds(req: Request): Response* ⇒ Calls the *getEquipmentIds(farmId: integer, fieldId: integer, cropfieldId?: integer, saveDate?: boolean)* function from the *Equipment Information Provider* class where it gets all parameters from req.body. It returns a Response object with in it an array of integers.
- *POSTlocationInformation(req: Request): void* ⇒ Calls the *postLocationInformatiopn(obj: Location Information)* function from the *Equipment Information Provider*

class with as obj a *Location Information* object that is created based on the values in req.body. This function does not return a response.

- *PUTlocationInformation(req: Request): void* ⇒ Calls the *updateLocationInformatiopn(obj: Location Information)* function from the *Equipment Information Provider* class with as obj a *Location Information* object that is created based on the values in req.body. This function does not return a response.
- *GETlocationInformation(req: Request): Response* ⇒ Calls the *getLocationInformation(equipmentId: integer)* function from the *Equipment Information Provider* class with as equipmentId req.query.id. It returns a Response object which contains a Location Information object in its body.
- *DELETElocationIformation(req: Reequest): void* ⇒ Calls the *deleteLocationInformation(equipmentId: integer)* function with as equipmentId req.query.id. This function does not return a response.
- *GETliveData(req: Request): Response* ⇒ Calls the *getLiveData(farmId: integer, fieldId: integer, cropfieldId?: integer)* function from the *Data Provider* class where it gets all parameters from req.body. It returns a Response object with in it an array of *Data Value* objects.

WolkyTolky Data Provider This class manages the WolkyTolky data that is stored for equipment and provides this data to the *Data Linking Controller* and the *Data Provider* classes. This class uses the *WolkyTolky Equipment* and *Location Point* classes to get and store its data.

functions

- *getEqupmentsWt(): Array<WolkyTolky Equipment>* ⇒ Retrieves all stored *WolkyTolky Equipment* objects and returns it in an array.
- *postEquipmentWt(obj: WolkyTolky Equipment): void* ⇒ Stores a $WolkyTolkyEquipment$ object in the database. This function does not return anything.
- *getEquipmentWt(id: integer): WolkyTolky Equipment* ⇒ Returns the $WolkyTolkyEquipment$ object that has as *equipmentId* the input parameter of the function, if there exists such an object.
- *putEquipmentWt(WolkyTolky Equipment): void* ⇒ Replaces the *WolkyTolky Equipment* object in the database that has the same *equipmentId* as the input object, with the input object. This function does not return anything.
- *deleteEquipment(id: integer): void* ⇒ Deletes the *WolkyTolky Equipment* object that has the input id as equipment id. Also calls the *deleteEquipmentInformation(equipmentId: integer)* with this id to delete the additional equipment information. This function does not return anything.
- *getWolkyTolkyStationGPS(apiKey: integer, stationId: integer): Array<Location Point>* ⇒ makes an API call to the WolkyTolky API and returns all available GPS locations of the specified station in an array of *Location Point* objects.

Equipment Information Provider This class manages the additional equipment information that is stored as $EquipmentInformation$ objects and provides this data to the *Data Linking Controller* and *Data Provider* classes.

functions

- *getEquipmentIds(farmId: integer, fieldId: integer, cropfieldId?: integer, saveDate?: boolean): Array<integer>)* ⇒ Returns the *equipmentId* of all equipment objects of which the parameter values match with the input values of this function. These are returned in an array of integers.
- *postEquipmentInformation(obj: Equipment Information): void* ⇒ Adds the input *Equipment Information* object to the database. This function does not return anything.
- *updateEquipmentInformation(obj: Equipment Information): void* ⇒ Replaces the *Equipment Information* object in the database that has the same *equipmentId* as the input object, with the input object. This function does not return anything.
- *getEquipmentInformation(equipmentId: integer): Location Information* ⇒ Returns the *Equipment Information* object with as *equipmentId* the given input, if it exists.
- *deleteEquipmentInformation(equipmentId: integer): void* ⇒ Deletes the object with the given *equipmentId*, if it exists. This function does not return anything.

Data Provider This class provides the front-end with requested measurement data from the equipment.

functions

- *getLiveData(farmId: integer, fieldId: integer, cropfieldId?: integer): Array<Data Value>* ⇒ Returns live sensor data from all equipment with certain parameters. First the *getEquipmentIds(farmId: integer, fieldId: integer, cropfieldId?: integer, saveDate?: boolean)* function from the *Equipment Information Provider* class is called with the input parameters to define for which equipment the data needs to be returned. After that the WolkyTolky API is called using the credential information that can be retrieved by calling the *getEquipmentWt(id:integer)* function of the *WolkyTolky Data Provider* class for every *equipmentId* that is retrieved before. After this all most recent data is parsed to *Date Value* objects and an array of these objects is returned.

WolkyTolky Equipment

parameters

- *equipmentId: integer* ⇒ Unique identefier for every instance of this class. The *equipmentId* corresponds with the *id* of the Equipment class used on the sensing API which is provided by the client.

- *apiKey: string* ⇒ The API key that is used to send requests to equipment running the WolkyTolky API.
- *stationId: integer* ⇒ The identifier of the piece of equipment on the WolkyTolky API.

functions

- *OnCreate(Props): void* ⇒ The function that is used to create a new instance of the Class.
- *getEquipmentId(): integer* ⇒ Returns the equipmentId of the object.
- *getApiKey(): string* ⇒ Returns the apiKey.
- *getStationId(): integer* ⇒ Returns the stationId

Location Point

parameters

- *latitude: float* ⇒ The latitude of a GPS location.
- *longitude: float* ⇒ The longitude of a GPS location.

functions

- *OnCreate(Props): void* ⇒ The function that is used to create a new instance of the Class.
- *getLatitude(): float* ⇒ Returns the latitude of the object.
- *getLongitude(): float* ⇒ Returns the longitude of the object.

Equipment Informatioin

parameters

- *equipmentId: integer* ⇒ Unique identefier for every instance of this class. The *equipmentId* corresponds with the *id* of the Equipment class used on the sensing API which is provided by the client.
- *farmId: integer* ⇒ The id of the farm where the piece of equipment is placed.
- *fieldId: integer* ⇒ The id of the field where the piece of equipment is placed.
- *cropfieldId?: integer* ⇒ The id of the cropfield where the piece of equipment is placed. This parameter does not have to have a value.
- *saveDate: boolean* ⇒ Stores whether the date and time of measured sensor values should be saved in the database.

functions

- *OnCreate(Props): void* ⇒ The function that is used to create a new instance of the Class.

- *getEquipmentId(): integer* ⇒ Returns the equipmentId of the object.
- *getFarmId(): integer* ⇒ Returns the farmId of the object.
- *getFieldId(): integer* ⇒ Returns the fieldId of the object.
- *getCropfieldId(): integer* ⇒ Returns the cropfieldId of the object, if the object has one.
- *getSaveDate(): boolean* ⇒ Returns if the date and time of measured sensor values should be saved in the database.

Data Value

parameters

- *name: string* ⇒ Stores the name of the data value.
- *unit: string* ⇒ Stores the unit of the value.
- *value: integer* ⇒ Stores the actual measured value.

functions

- *OnCreate(Props): void* ⇒ The function that is used to create a new instance of the Class.
- *getName(): string* ⇒ Returns the name of the object.
- *getUnit(): string* ⇒ Returns the unit of the object.
- *getValue(): integer* ⇒ Returns the value of the object.

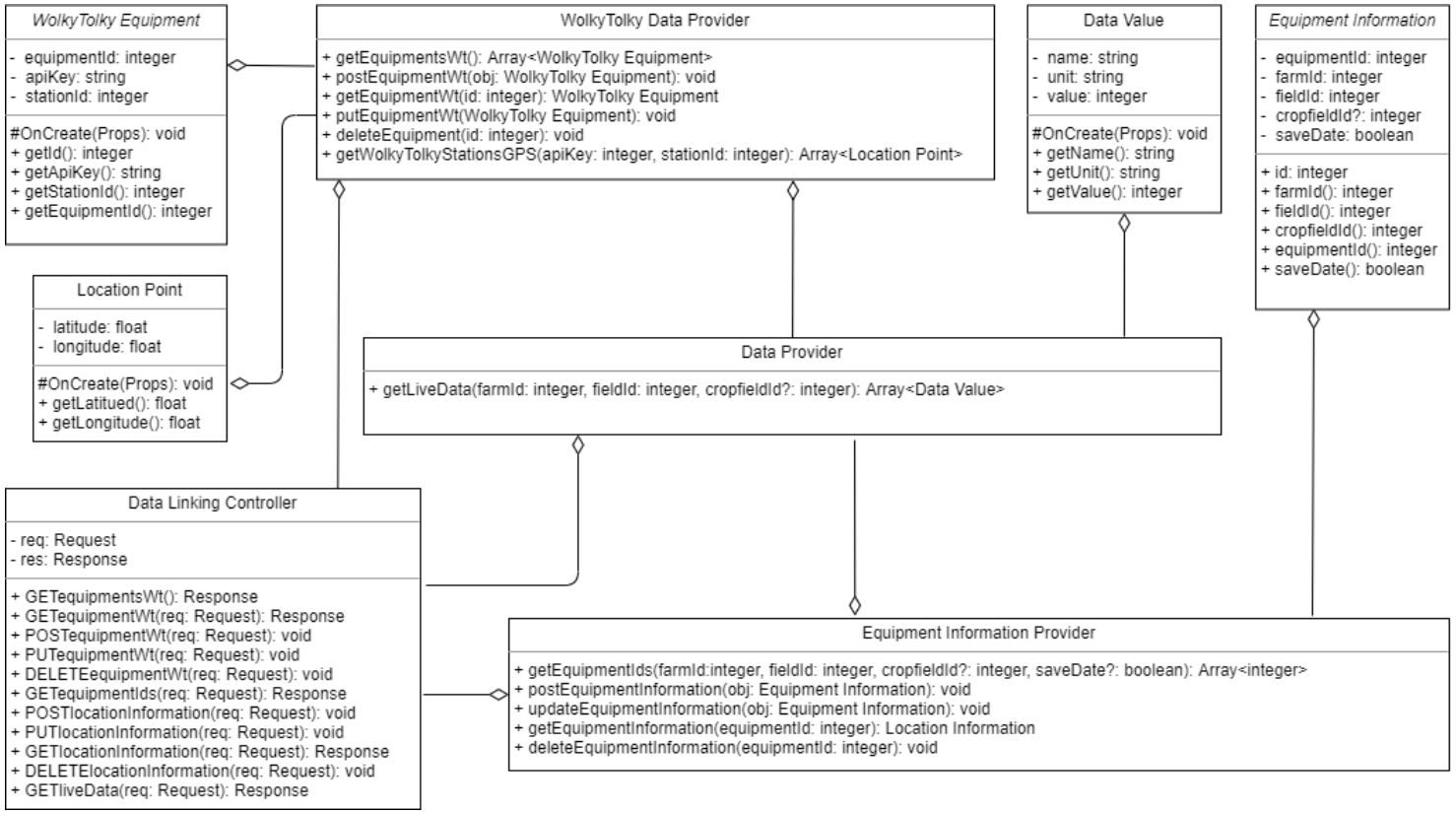

Figure 8: UML class diagram Data Linking API.

3.3 State dynamics

This section will illustrate the dynamic behaviour of the system by the use sequence diagrams. Each diagram is accompanied with a textual explanation.

3.3.1 Create an account

If a user wants to use the application, they first need to create an account to be uniquely identifiable. This is done by clicking on the sign-up link, which takes you to the sign up page. On the sign up page, a user needs to fill in its first name, last name, email, password, and password confirmation. Firstname, lastname, email, password, and confirm password are all required fields to fill in, the password and the confirm password need to be the equal, and the password need to be at least 5 characters. The database also checks if the email already exists, which, if true, returns an error message to inform the use to choose a different email.

Goal
Create a new account.

Precondition
User has accessed the login view.

Postcondition
A new account is created and the user is logged in.

User
All users.

Summary
A new user can create an account by clicking the sign up link and filling in their credentials. If the credentials are valid a new account will be created, otherwise the user will be redirected to the login view.

Priority
Must have

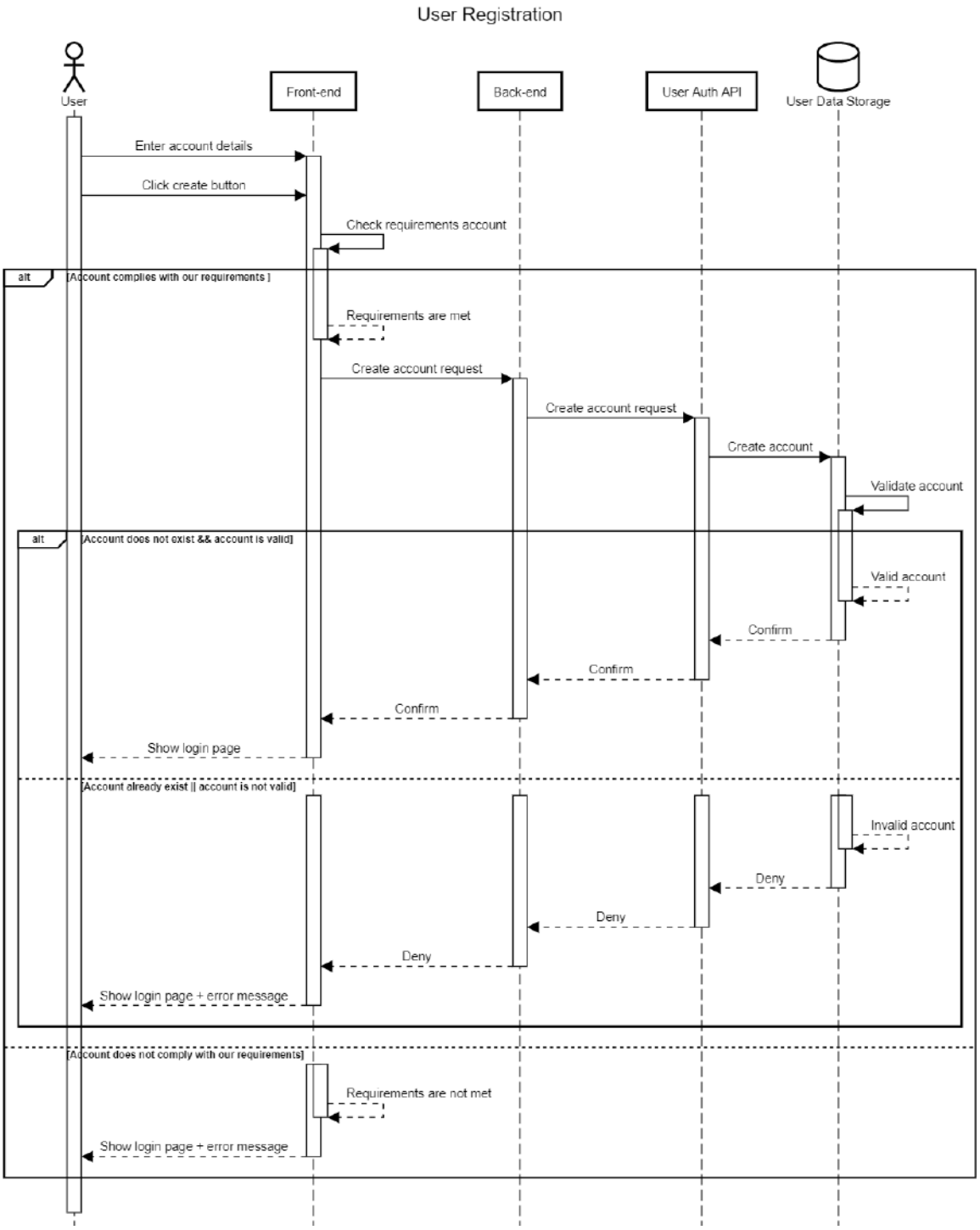

Figure 9: Create new account sequence diagram

3.3.2 User login

To allow the users to use the application, they first need to log in. This is because the rights people have on a farm are user-specific, so they need to identify themselves. This is done by having users log in to the website with their email and their password.

Goal
A user that wants to enters the application needs to enter their email and password in order to get access.

Precondition
The user has an account on CloudFarmer.

Postcondition
If the credentials are correct the user is logged in and gets redirected to the home page

User
All users.

Summary
The user enters the web application to fill in its credentials. If the credentials are correct the user will login and be directed to the home page. If the credentials are not correct the user will get an error message.

Priority
Must have

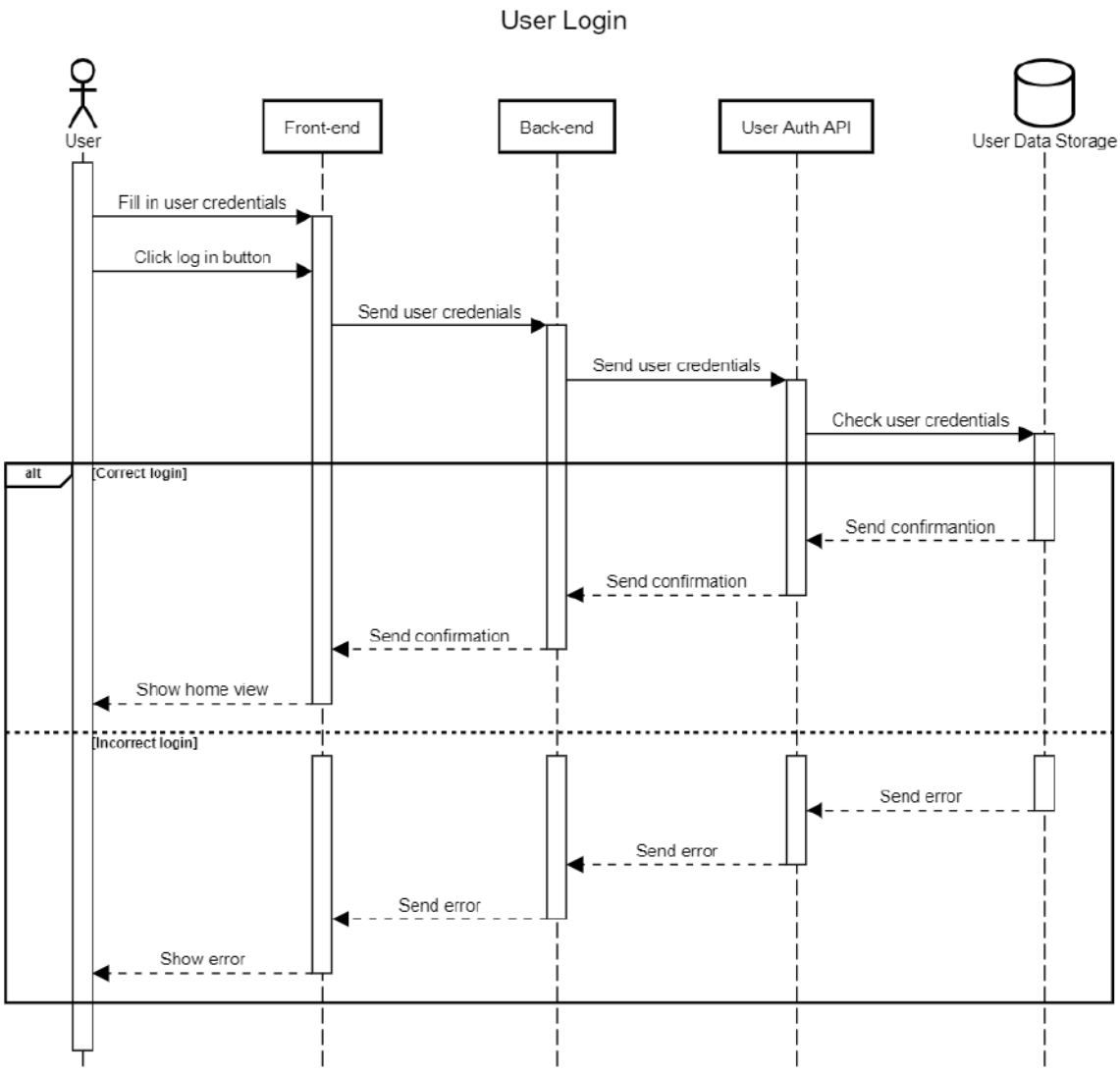

Figure 10: User login Sequence Diagram

3.3.3 User Logout

After users are logged in to the website they also need an option to logout. This is standard functionality for the website to keep the data more save.

Goal
The goal is that a user that has logged into the application can logout of the application.

Precondition
The user is logged in.

Postcondition
The user is logged out and redirected to the login page.

User
All users.

Summary
A user can logout of the system by clicking the logout button. After clicking the button the user is redirected to the login page.

Priority
Must have

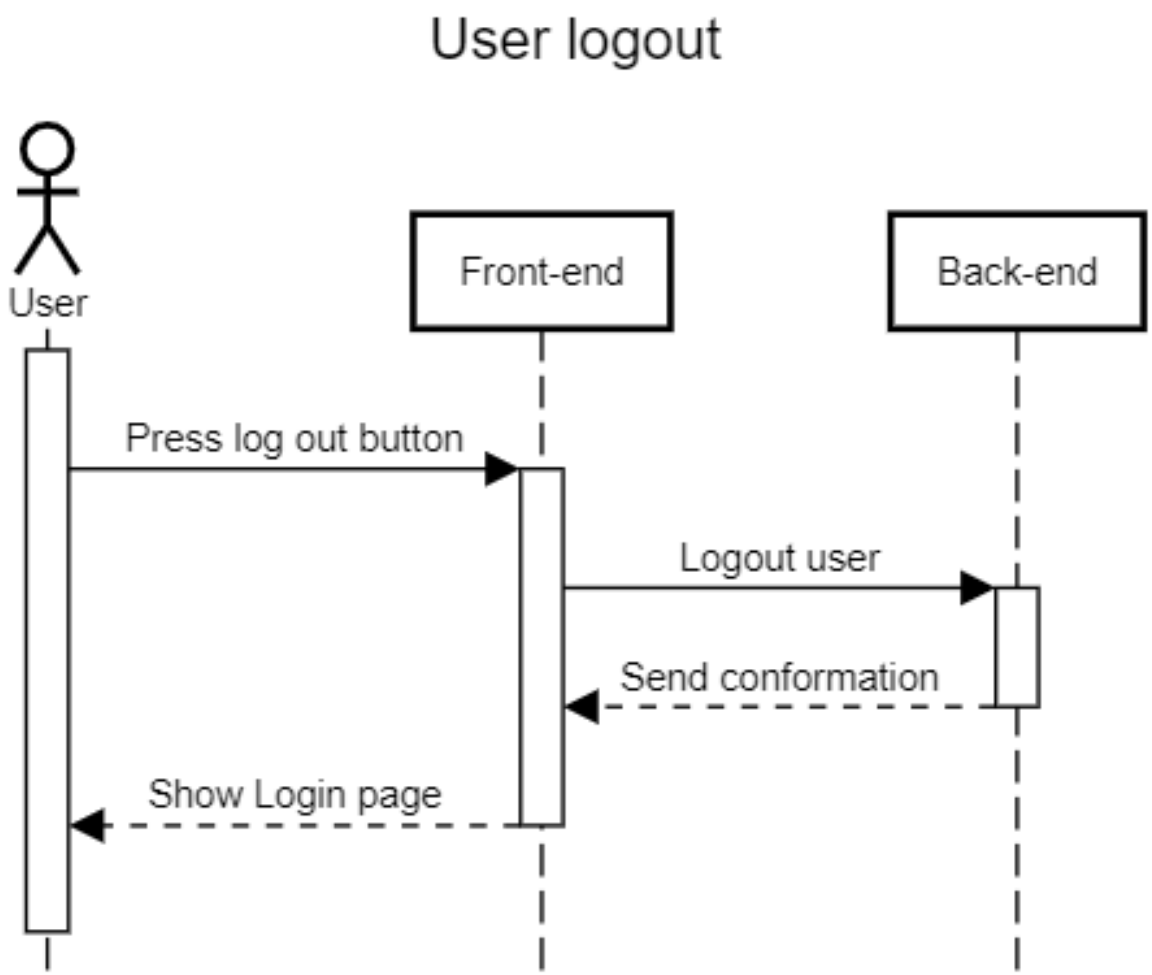

Figure 11: User logout Sequence Diagram

3.3.4 Edit personal settings

A user can change their previous set personal settings, such as their first name, last name or email.

Goal
The goal is that a user that is logged in can change their personal settings.

Precondition
The user is logged in and on the personal settings page.

Postcondition
The settings are edited to the preference of the user.

User
All users.

Summary
A user edits the personal settings on the personal settings page.

Priority
Must have

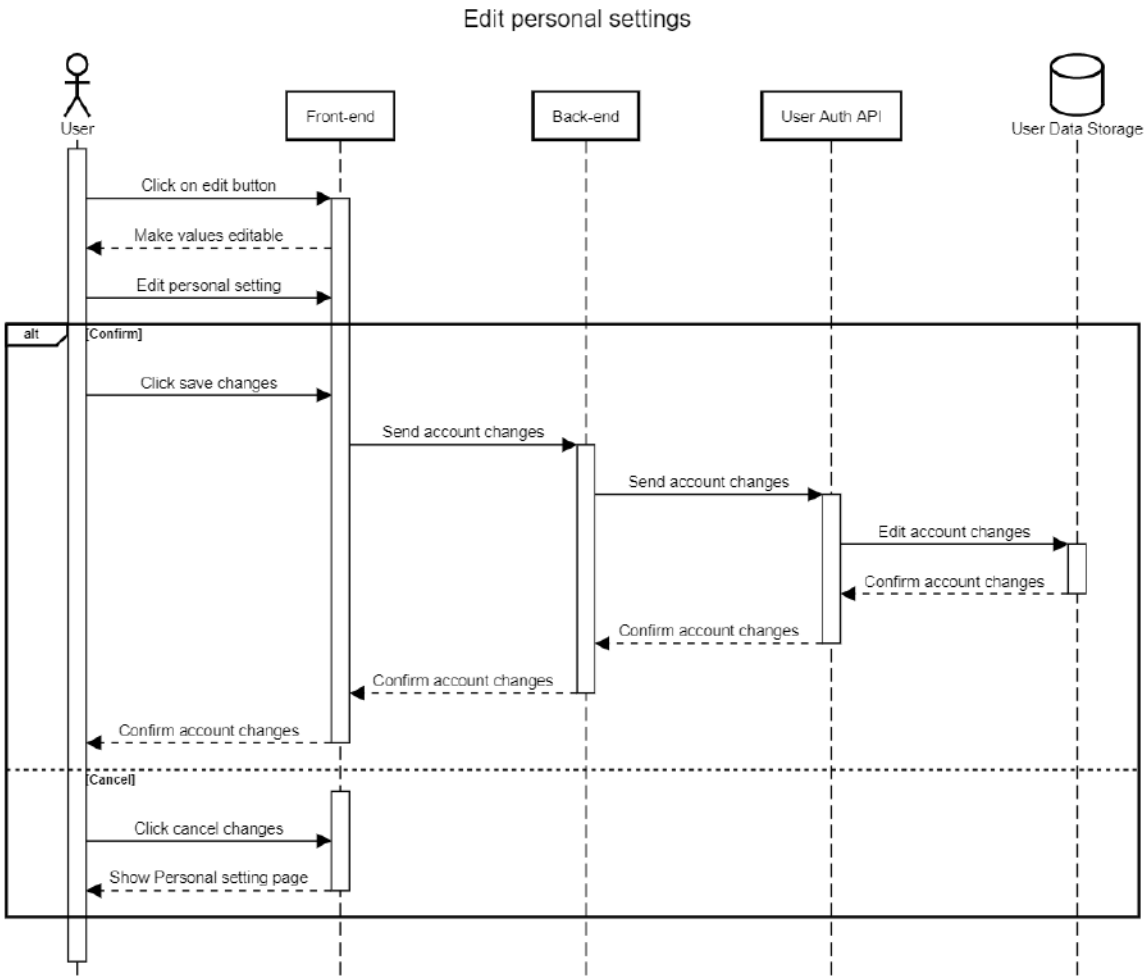

Figure 12: Edit Personal Settings Sequence Diagram

3.3.5 Delete account

If a user doesn't want to use the application anymore they can delete their account. This can be done by going to the personal setting page, on this page they will find a delete my account button. Upon pressing this button they get asked for confirmation if they want to delete the account or not. If they cancel the confirmation the account will keep existing.

Goal
Delete an existing account.

Precondition
A user is logged in, and the user is on the home view.

Postcondition
The account of the user will be deleted.

User
All users.

Summary
A user can delete their own account. By going to personal settings and deleting their account via the delete button.

Priority
Should have

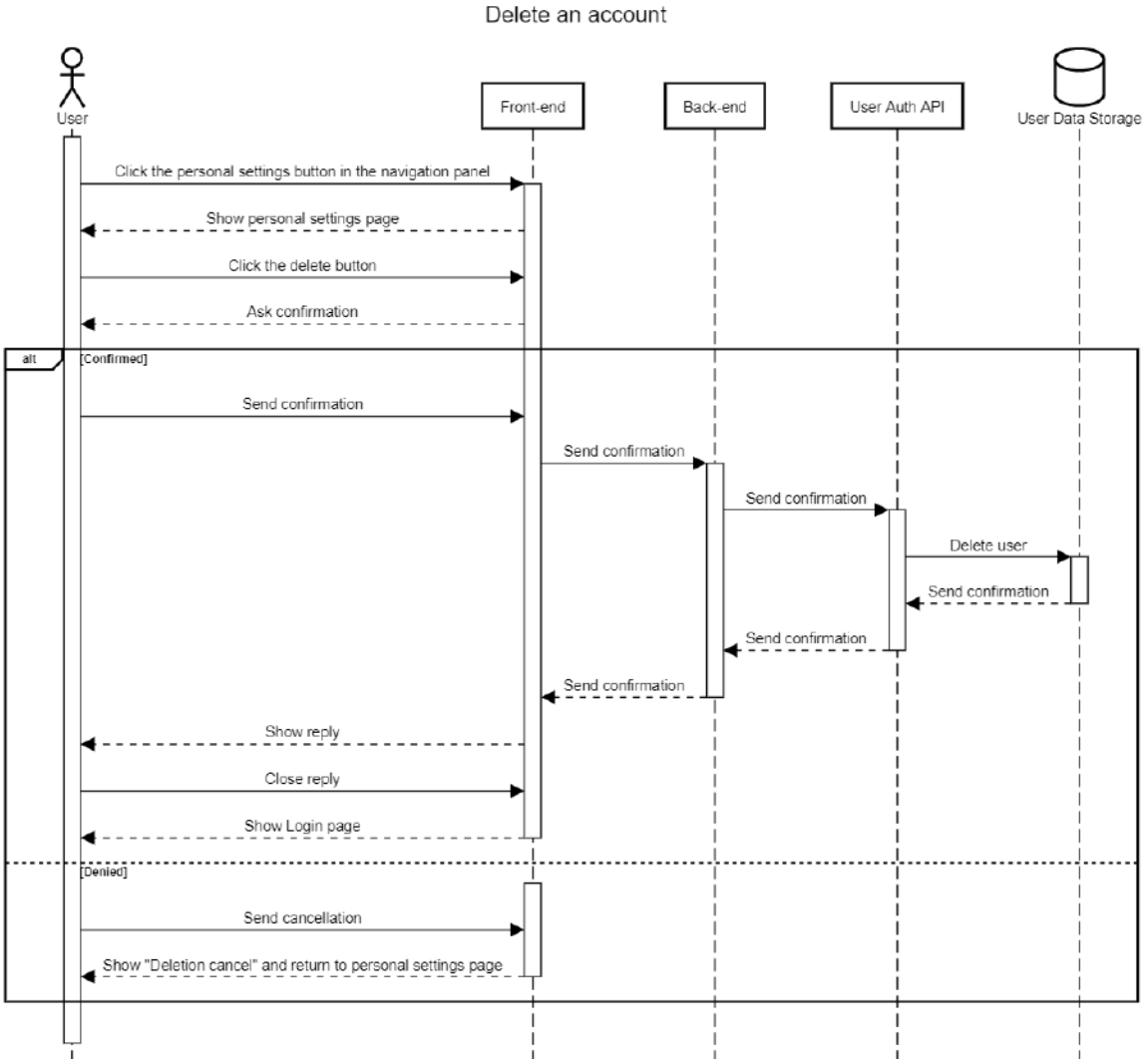

Figure 13: Delete Account Sequence Diagram

3.3.6 Change password

Once a user has an account they might want to change their password. To do this the user has to go to the personal settings page. On this page they will find the change password button. When they click on this button they can change their password. This is done by entering their password and creating a new password and confirming the new password.

Goal
Changing the old password to a new and valid password.

Precondition
A user is logged in, and is on the personal settings view.

Postcondition
Password is changed.

User
All users.

Summary
A user changes their password by entering their old password and entering a new password and confirming the new password.

Priority
Should have

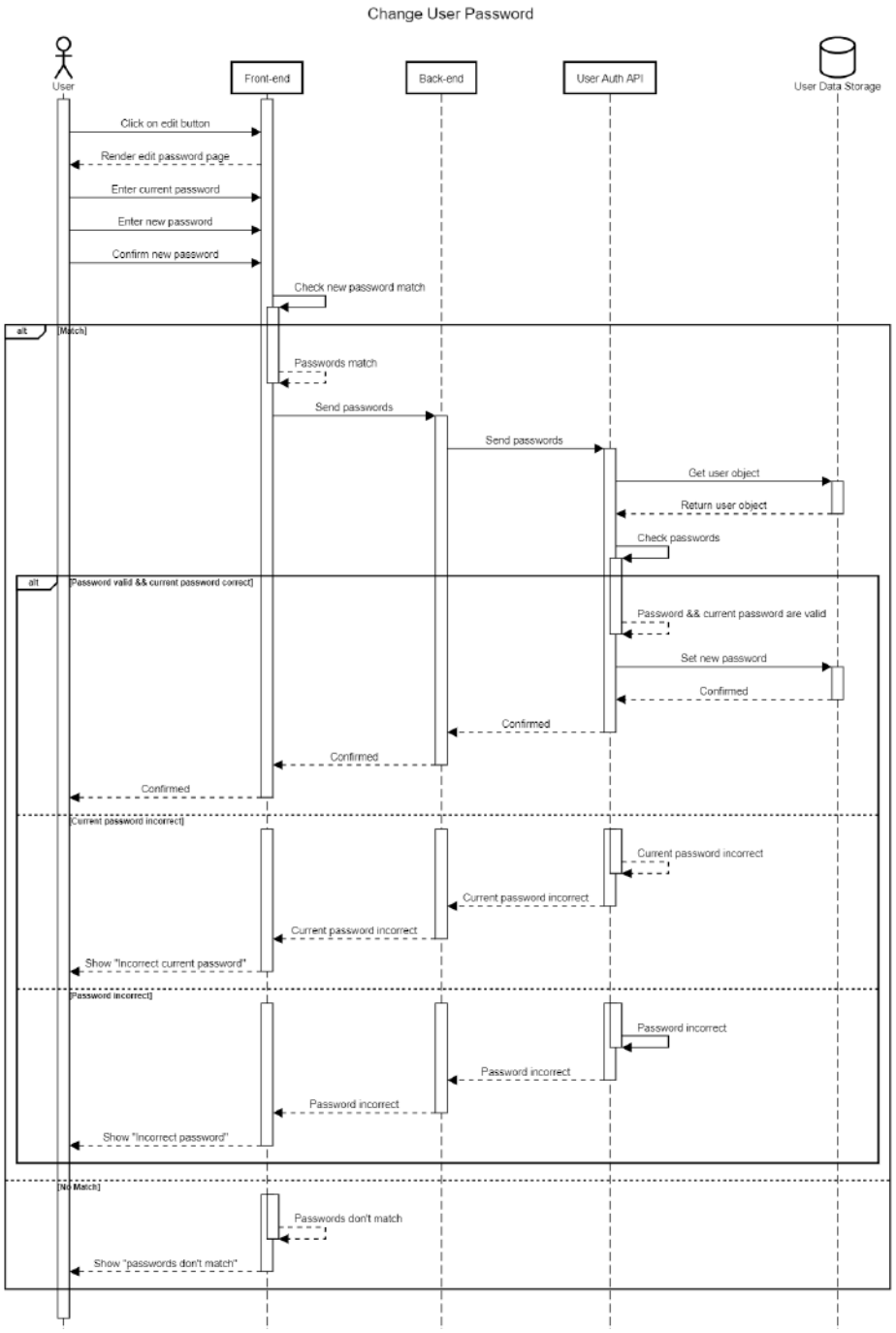

Figure 14: Change user password Sequence Diagram

3.3.7 User forgets password

Once a user is registered the user might forget their password. The user wants to reset their password. On the login page, the user can click the reset password button. Which will redirect to a page where the email of the user can be filled in. After the email is filled in the user can press the send button to send a password reset link to the filled-in email address. After the user has clicked the link in the received email. The user will be shown a password reset form. If filled in correctly the password will be changed to the newly chosen password and the user will be redirected to the login page.

Goal
The user is on the login page, the user doesn't know its password.

Precondition
The password of the user is reset.

Postcondition
All users.

User
A user forgets their password and resets their password.

Summary
Must have

Priority

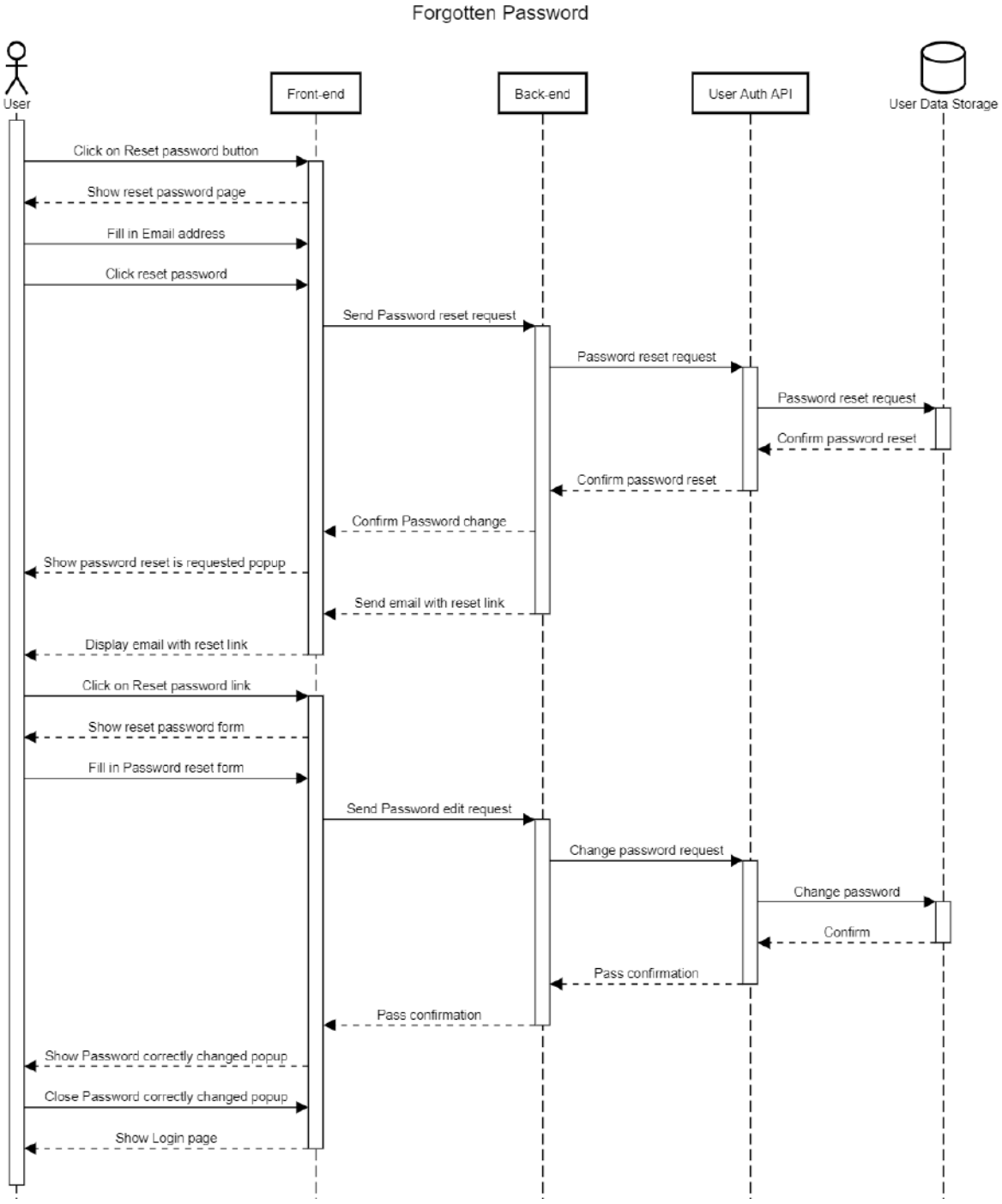

Figure 15: User forgets password

3.3.8 Create a farm

Once a user is registered the user might want to create a farm. Once the user is on the home page there will be a create a farm button. Once they click on this button they get provided with a popup dialogue, where the farm information can be filled in. This is done by filling in all the required fields and then pressing the create a new farm button, which then redirects you to the home page. The creation can also be cancelled by clicking the cancel button, which then also redirects you to the home page.

Goal
Create a farm.

Precondition
A user is logged in, and the user is on the home view.

Postcondition
A farm will be created.

User
All users.

Summary
A user that wants to create a farm can create a farm.

Priority
Should have

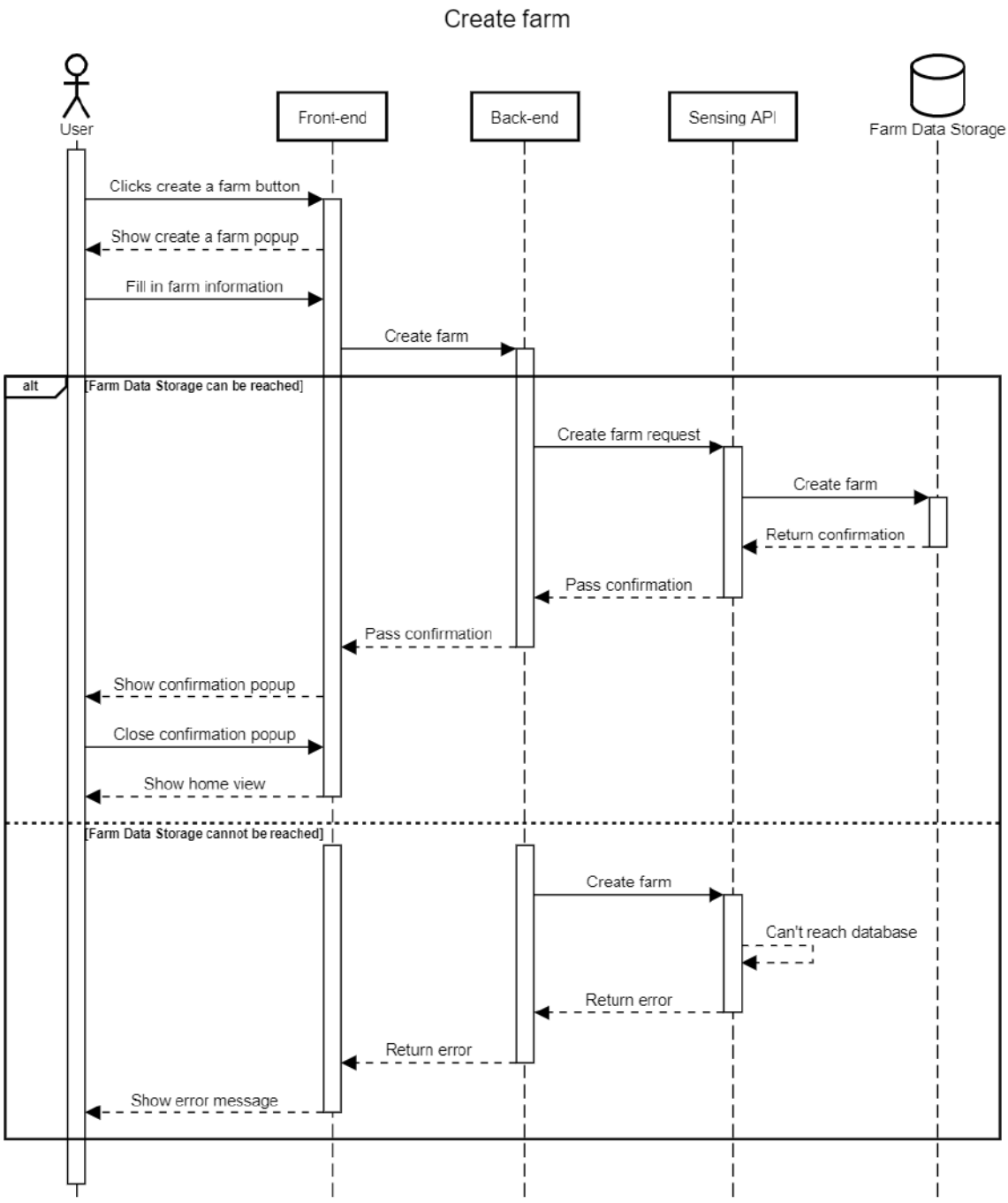

Figure 16: Create a farm Sequence Diagram

3.3.9 Get field and crop field information

If the users have the correct roles/rights they can view field and crop field information. To do this they have to select the field view. On this view, they can see all the fields that belong to the active farm. They can select a farm and get more information about that field. After they have selected a field they can select a crop field inside of the field, this way they get the crop field information.

Goal

To get information about the fields and/or crop fields of a farm.

Precondition

The user is logged in, and has rights to see the fields and crop fields of a farm. This means a user is a researcher, farmer, or farm admin. Or the farm information is set to public. The user is on the home view.

Postcondition

User has information about the field and/or crop field of a farm.

User

Farm admin, Farmer, Researcher, and General User if farm is set to public.

Summary

User gets information about the field and/or crop field of a farm.

Priority

Must have

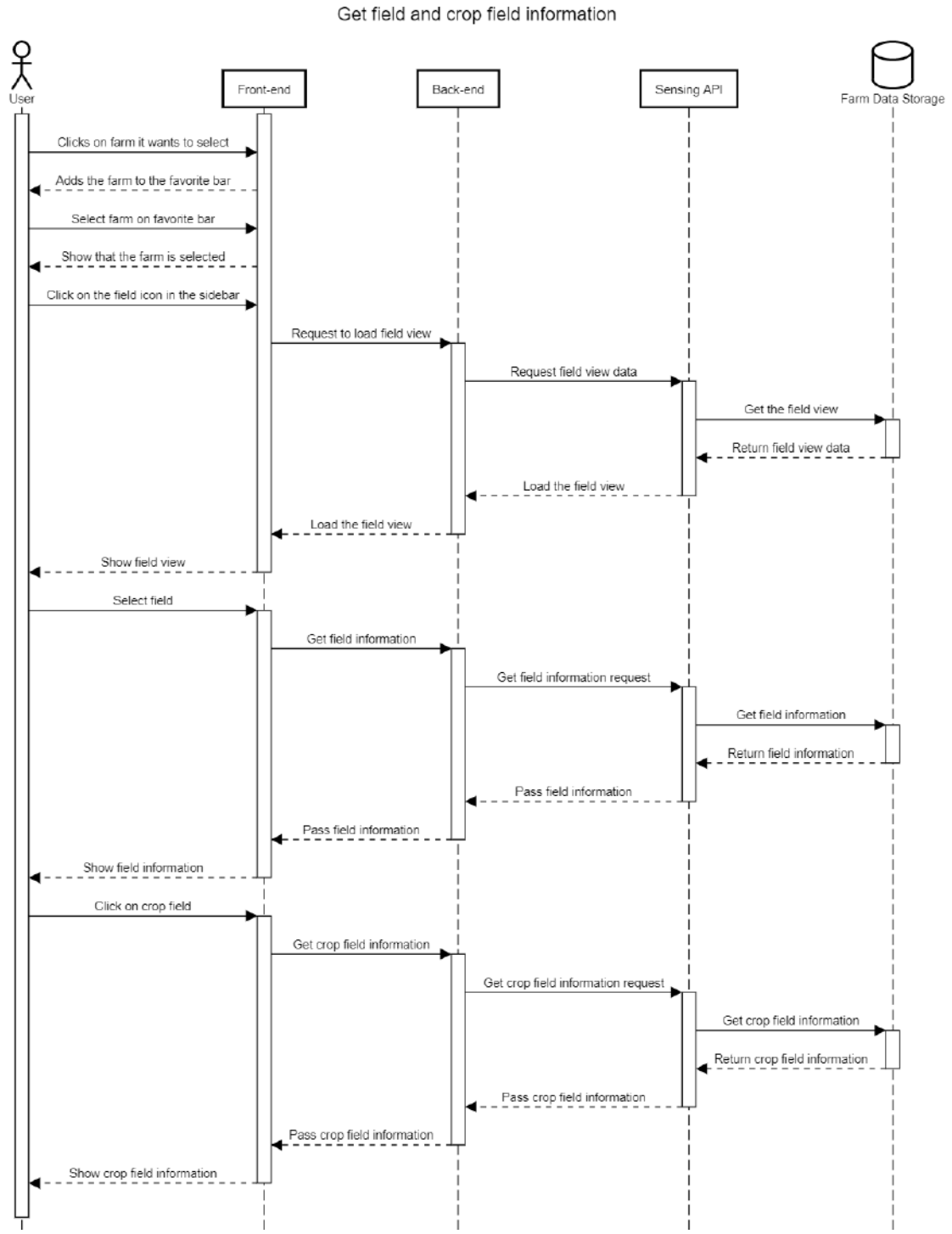

Figure 17: Get field and crop field information Sequence Diagram

3.3.10 Get observation data

Users want to see data that has been collected by the sensors. This way users can see what is happening on the farm or what happened on the farm. This can be done by going to the live view and going to the history view.

Goal

Get the measured data from the sensors on the farm.

Precondition

User is logged in, they are on the home view, and has the rights to see observation data. This means the user is a researcher, farmer, farm admin, or the farm admin has set the farm information to public.

Postcondition

User found data that has been measured at the farm fields.

User

Farm admin, Farmer, Researcher, and General user if the farm is set to public.

Summary

The user either goes to the history view or the live view, here they can see data from the active farm.

Priority

Must have

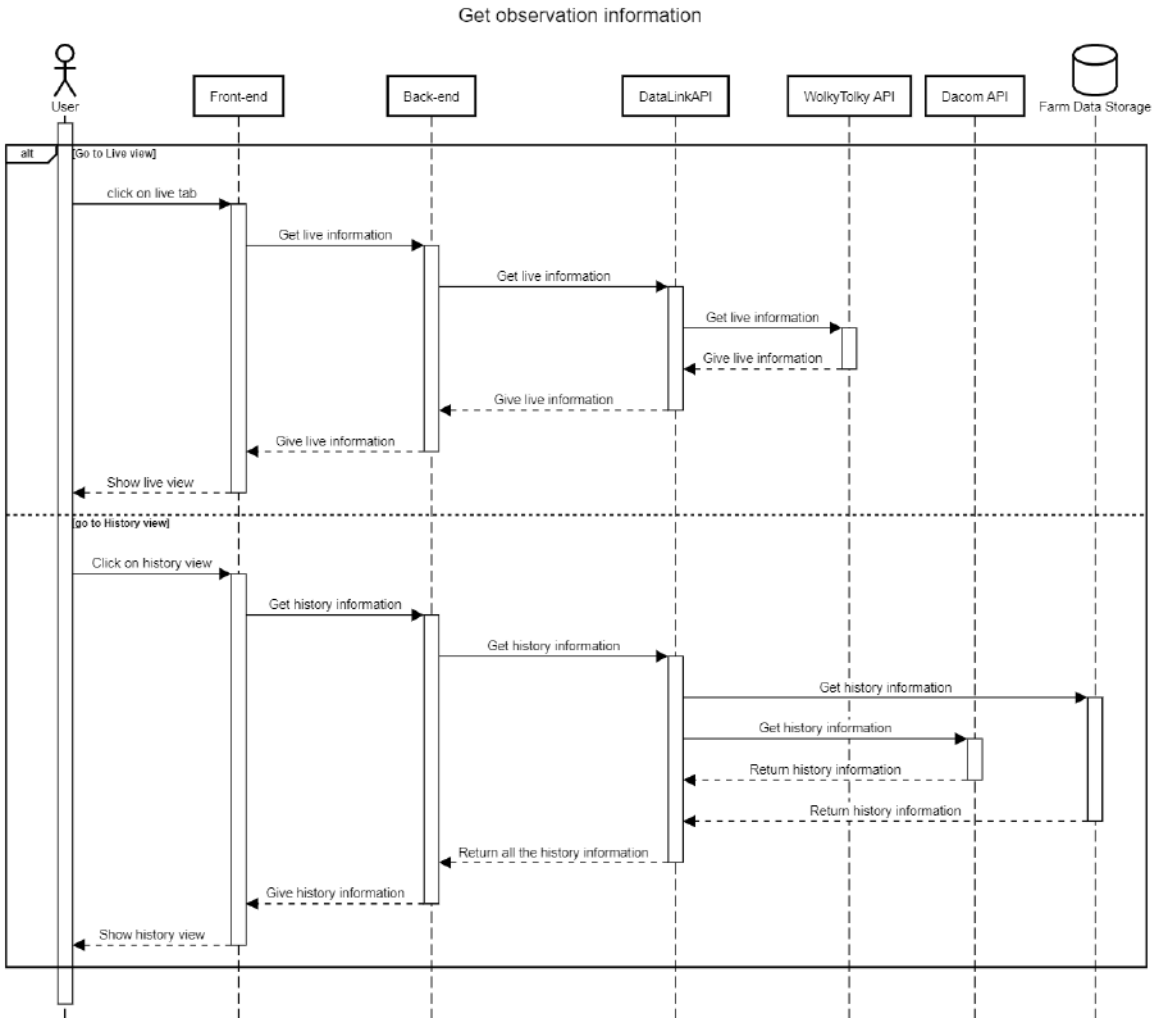

Figure 18: Get observation data Sequence Diagram

3.3.11 Get equipment data

To see what sort of sensors are being used on a farm, a user can go to the equipment view. This view gives an overview of what sensors are used and the information about sensors such as model, brand, etc.

Goal

To find out what sensors are on a farm and what their function is.

Precondition

A user that wants to see equipment data should be logged in and they should have one of the following roles researcher, farmer, or farm admin. If they are a general user the farm admin needs to set their farm information to public.

Postcondition

Users knows what sensors are on a farm and what their function is.

User

Farm admin, Farmer, Researcher, and General User if the farm is set to public.

Summary

The users looks at the equipment view and gets to know what sensors there are and what their function is.

Priority

Must have

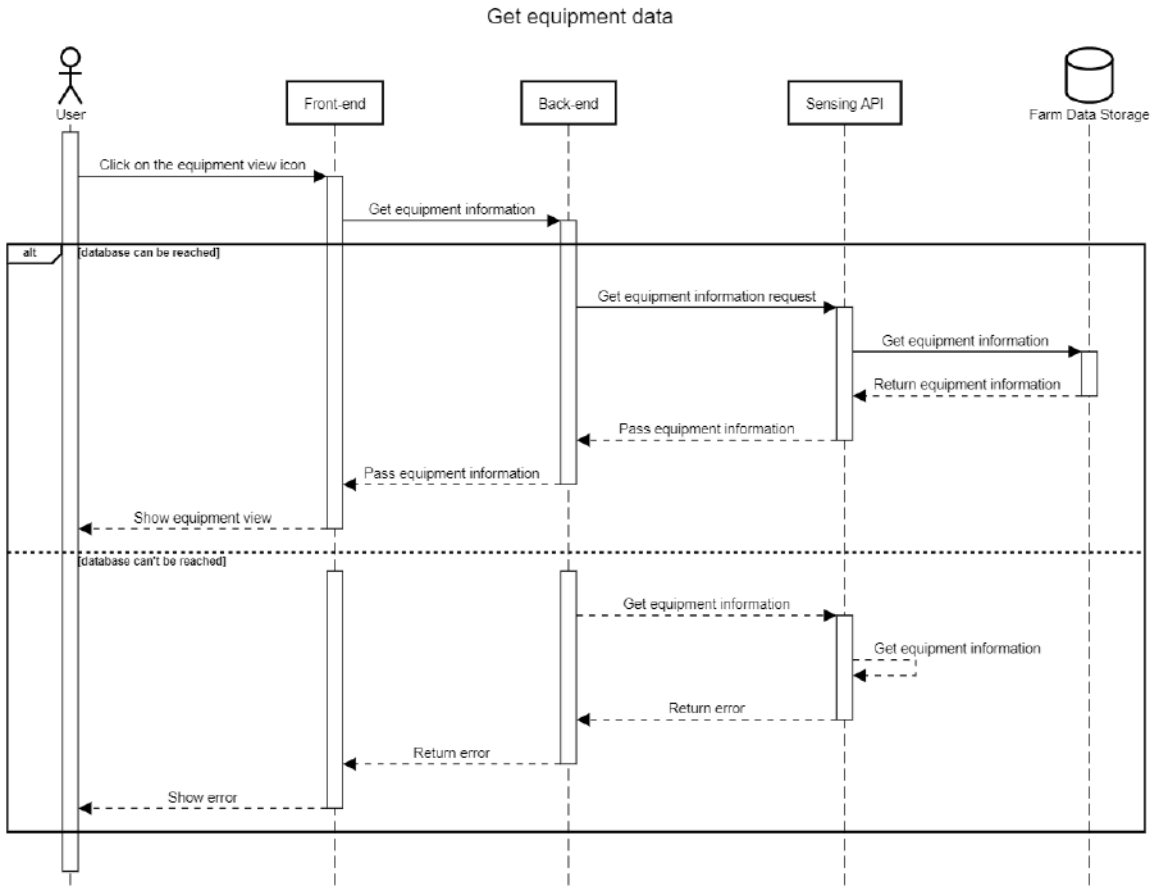

Figure 19: Get equipment data Sequence Diagram

3.3.12 Delete Field

A logged-in user might opt to delete a certain field's instance from the database. This removes its associated crop fields as well. The user navigates to the fields list in the fields view, selects the field they want to discard and then confirms its deletion.

Goal

Deleting a field's instance from the database.

Precondition

The user is logged in, the user is on the fields view, and the user is authorised to delete a field. This means that the user is a farm admin or a farmer.

Postcondition

The field is deleted.

User

Farm admin and Farmer.

Summary

The user deletes a field from the fields view.

Priority

Must have

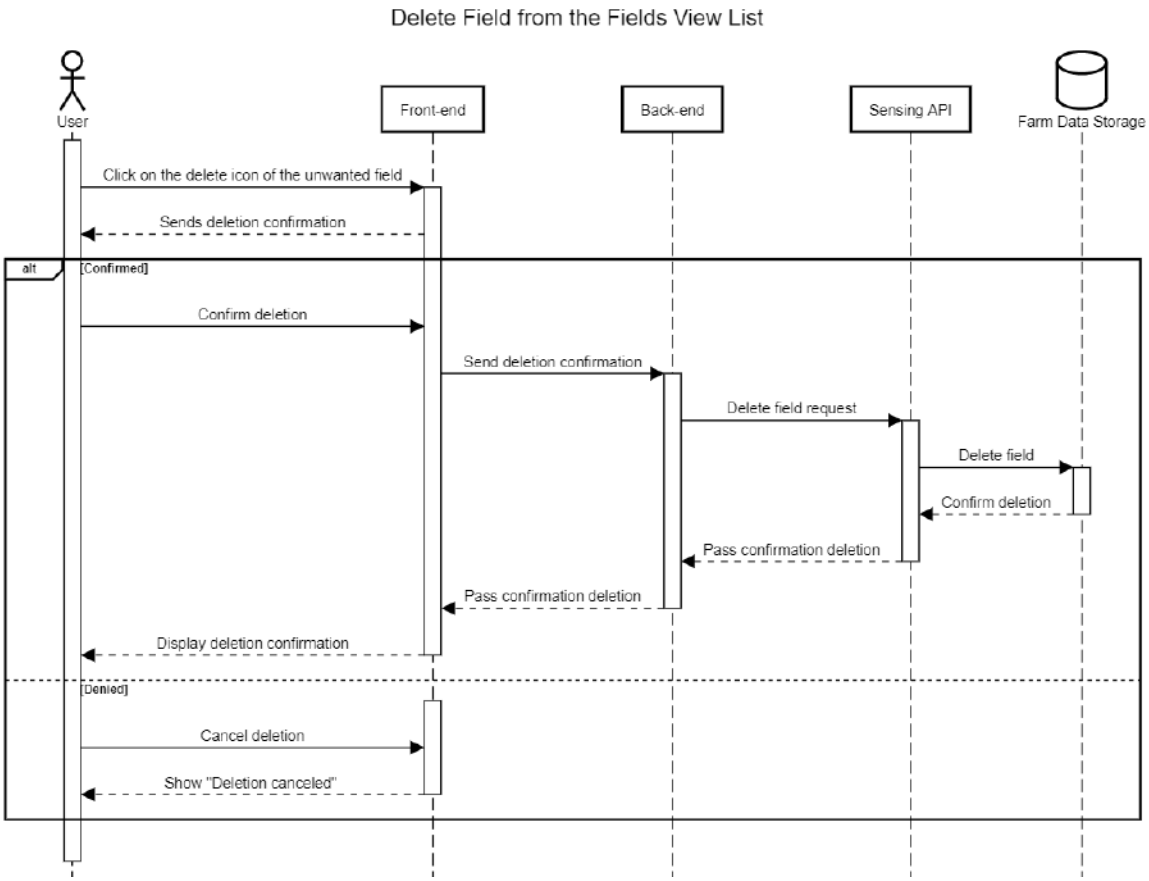

Figure 20: Delete Field from Fields View Sequence Diagram

3.3.13 Delete Observation Data

A logged-in user might opt to delete a certain farm's observation data. Assuming the user is at the history view and has permission, the user can navigate to the data tuple of the farm they want to delete and then clicks on its associated delete icon. The user then receives a message to confirm whether they want the data tuple to be removed or not.

Goal
Deleting a farm's observation data.

Precondition
The user is logged in, the user is on the history view, and the user is authorised so delete observation data. This means that the user is a farm admin, farmer, or researcher.

Postcondition
The farm's observation data is deleted.

User
Farm admin, Farmer, and Researcher.

Summary
The user deletes a farm's observation data.

Priority
Must have

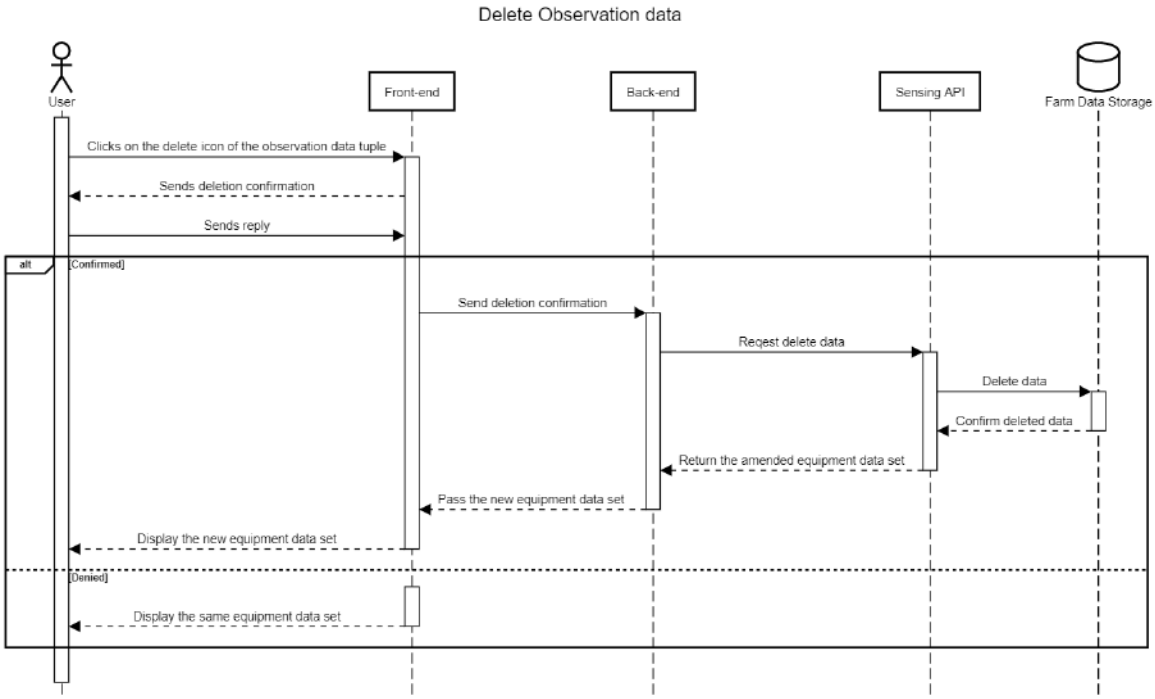

Figure 21: Delete Observation Data Sequence Diagram

3.3.14 Delete Equipment Data

If the user is authorised, they can delete equipment data of the active farm when they are in the Equipment View. The user can navigate to the data tuple of the piece of equipment they want to erase and then clicks on its associated delete icon in the equipment table.

Goal
Deleting equipment data.

Precondition
The user is logged in, the user is on the equipment view, and the user is authorised to delete equipment data. This means that the user is a farm admin or a farmer.

Postcondition
The equipment data is deleted.

User
Farm admin and Farmer.

Summary
The user deletes a farm's equipment data.

Priority
Must have

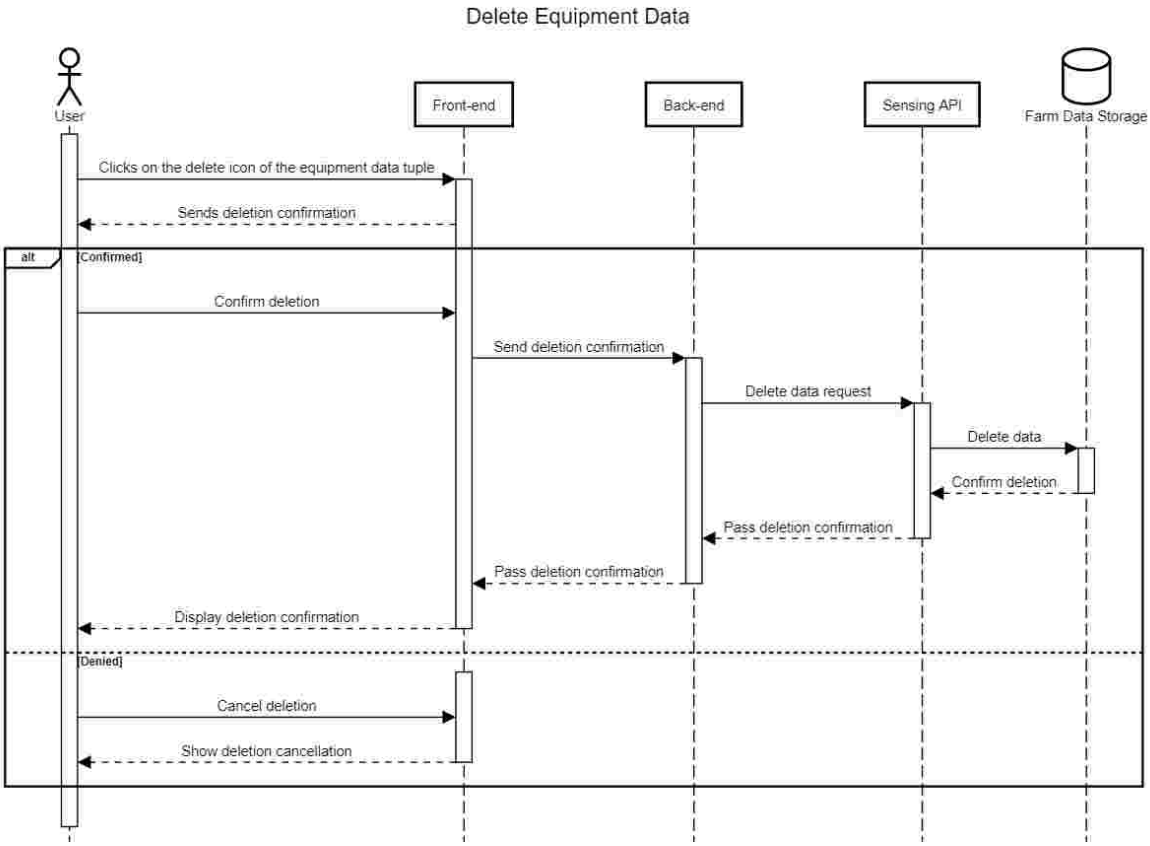

Figure 22: Delete Equipment Data Sequence Diagram

3.3.15 Delete Crop Field

A logged-in user might opt to delete a certain crop field's instance from the database. The user navigates to the fields list in the fields view, selects the crop field they want to discard and then confirms its deletion.

Goal

Deleting a crop field's instance from the database.

Precondition

The user is logged in, the user is on the equipment view, and the user is authorised to delete equipment data. This means that the user is a farm admin or a farmer.

Postcondition

The crop field is deleted.

User

Farm admin and Farmer.

Summary

The user deletes a crop field from the fields view.

Priority

Must have

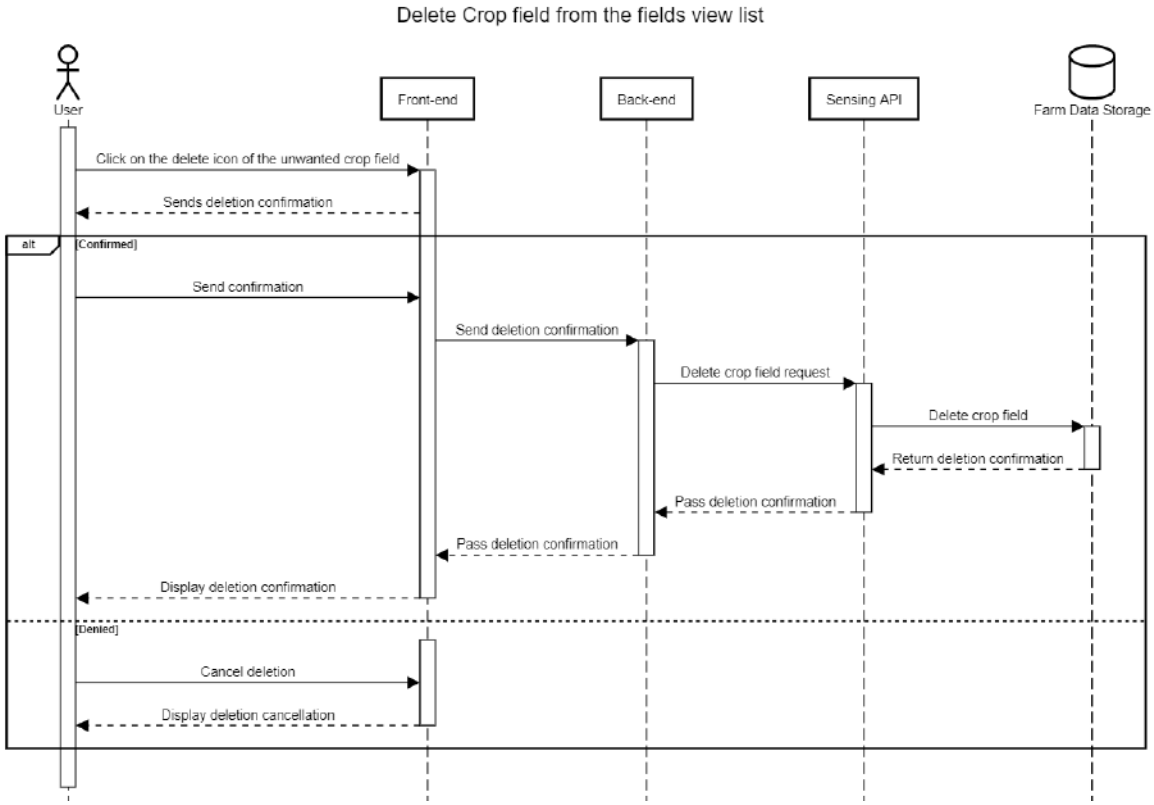

Figure 23: Delete Crop Field from Fields View Sequence Diagram

3.3.16 Filter Data

A logged-in user might opt to search for specific information in a certain view. Therefore, CloudFarmer supports filtering data in the Field View, History View and the Live View. Once the user is at the view they desire, they choose the desired filter header and type the information they want to be displayed.

Goal
Filtering data.

Precondition
The user is logged in and is on their desired view.

Postcondition
The data is filtered and displayed.

User
All users.

Summary
The user searches for the information they want and it gets displayed if it is in the database.

Priority
Should have

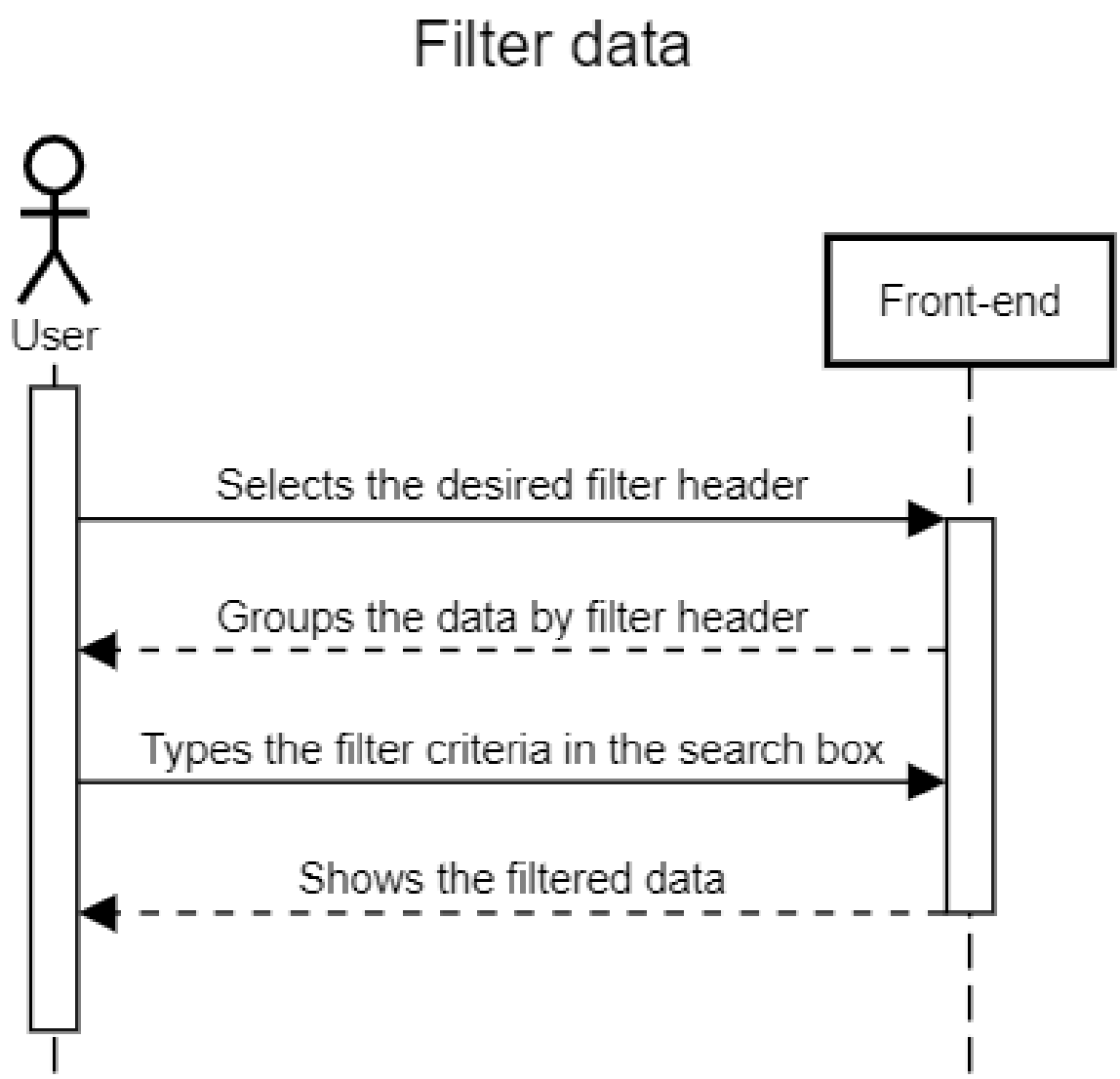

Figure 24: Filter Data Sequence Diagram

3.3.17 Upload CSV file

Some of the users wont have their sensors connected to the internet so the data won't get to the application through the API. They need to upload data by hand by uploading a CSV file.

Goal

Upload a CSV file and be able to see the data inside of a table.

Precondition

The user should be a researcher, farmer, or farm admin. The user should be logged in and on the history view.

Postcondition

The CSV file is uploaded and the data is visible to the user.

User

Farm admin, Farmer, Researcher, and General User when the farm is set to public.

Summary

A user uploads a CSV file and gets to see the data in a clear way.

Priority

Could have

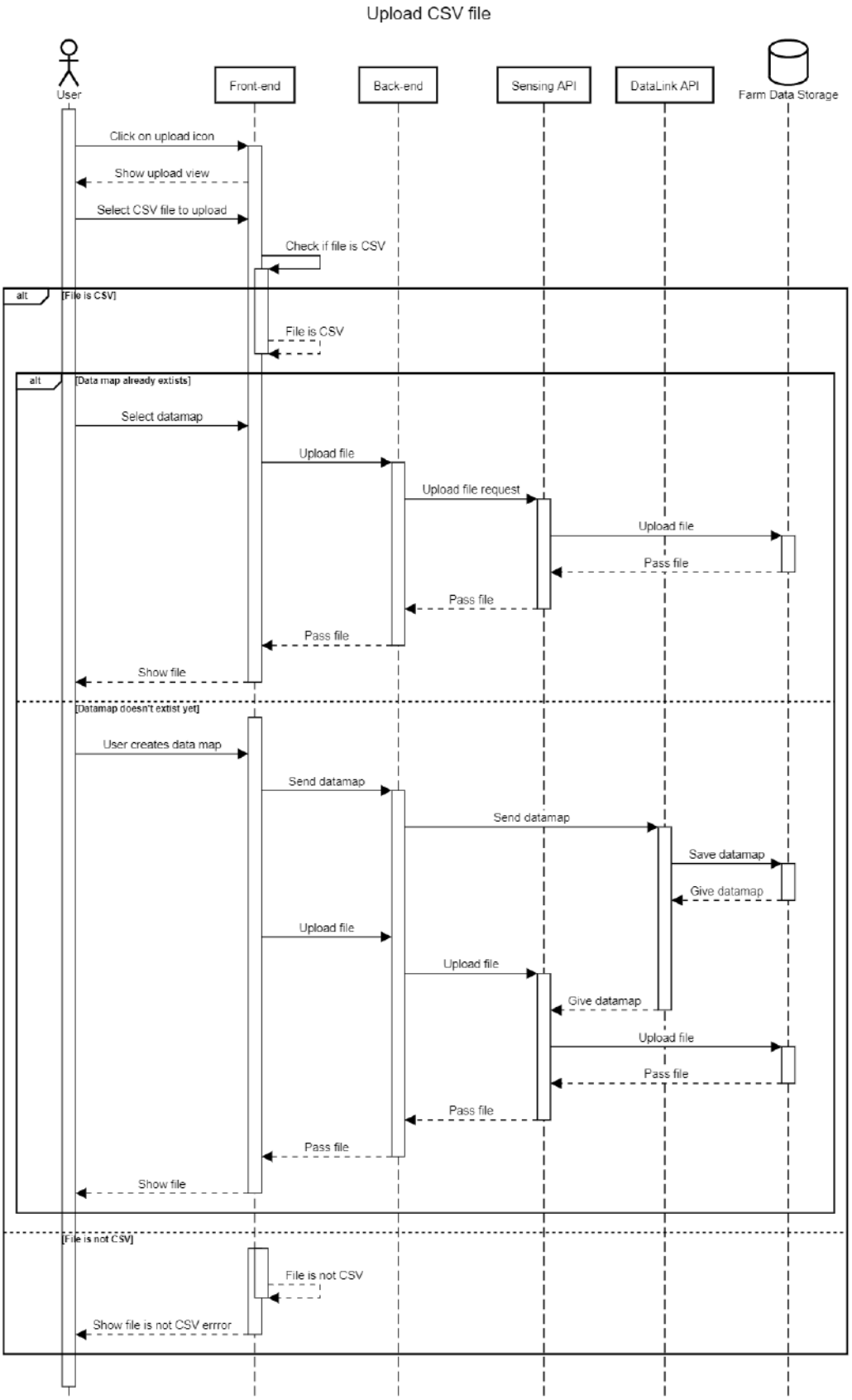

Figure 25: Upload CSV file Sequence Diagram

3.3.18 Add/Edit/Delete farm data

Farm admins can add, edit, and delete farm data. This is because there can be new data available for farm data, but also the data that is given can be outdated so it needs to be edited. Also, some data can be unnecessary and can be deleted from the farm data.

Goal
To add, edit, or delete farm data.

Precondition
The user should be a farm admin. The user should be logged in.

Postcondition
Farm data is added, edited, or deleted.

User
Farm admin.

Summary
A farm admin can add, edit, or delete farm data by going to the farm setting page and click on the edit button. Now they can add, edit, or delete data.

Priority
Should have

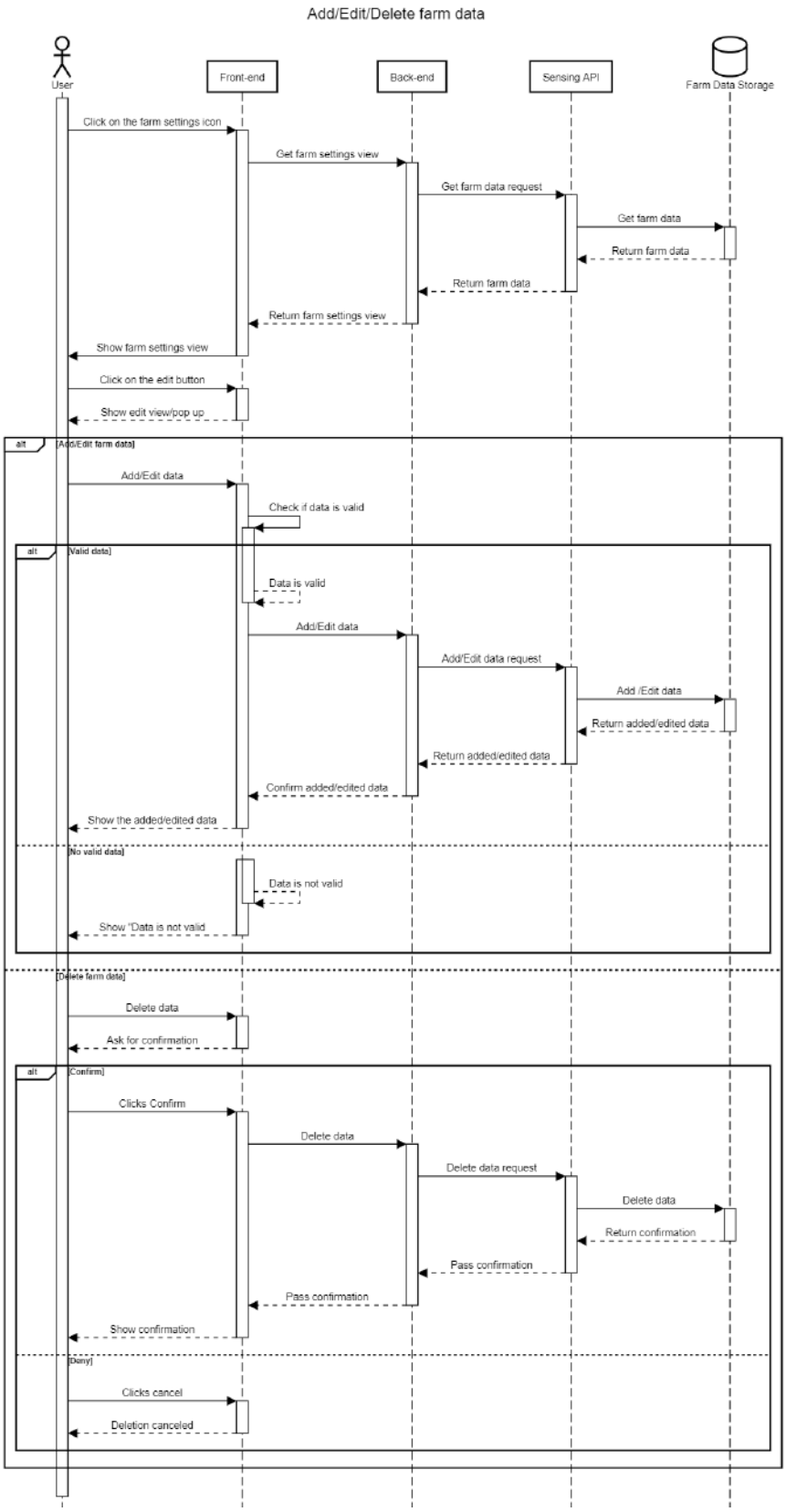

Figure 26: Add/Edit/Delete farm data

3.3.19 Edit field/crop field data

The farm fields can change during the lifetime of a farm. To make sure the application can handle this the fields can be edited in shape. Crop fields will change more often than fields since each season the crop fields can be redesigned. This is why the crop fields are also editable, in shape and the number of crop fields.

Goal

To edit fields and crop fields to match the current situation.

Precondition

A user is a farmer or a farm admin. The user is logged in and is on the home view.

Postcondition

The field/crop fields are changed in size or amount of fields/crop fields.

User

Farm admin and Farmer.

Summary

A farmer or farm admin can edit the fields and crop fields on the farm. This way the current situations is represented and the all users that can see the data know how the fields look.

Priority

Should have

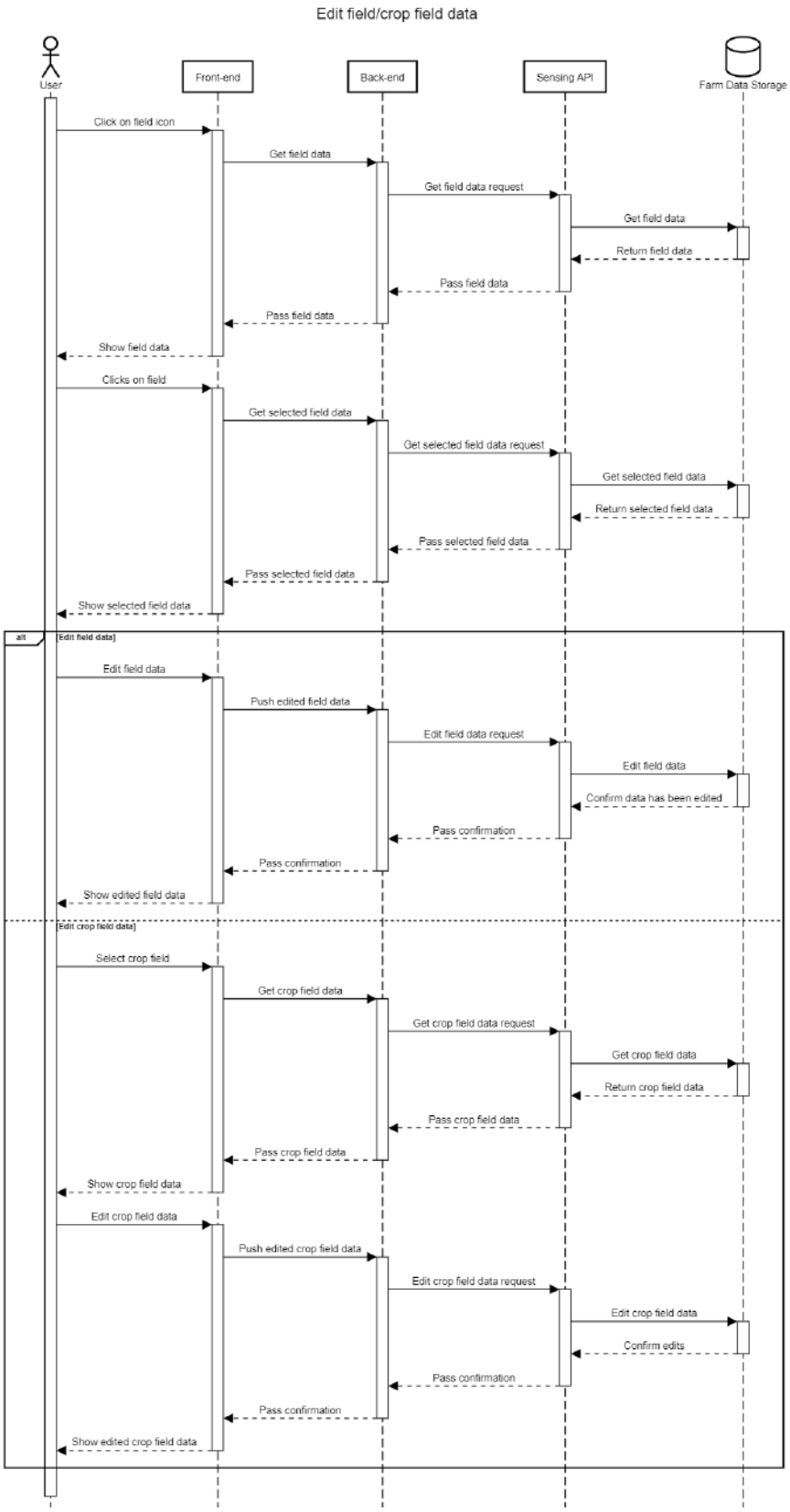

Figure 27: Edit Field/Crop field data Sequence Diagram

3.3.20 Change equipment data

The sensors that are on a farm can change a lot because sensors can be replaced, added, or broken down. To make sure the application can keep up with this farm admins and farmers can edit this data to make sure the sensors in the field are on the application.

Goal

To edit the equipment data, this can be adding, deleting, or edit the equipment data.

Precondition

The user is either a farm admin or a farmer, they are logged in. And they are on the home view.

Postcondition

The data of the equipment of the active farm is changed.

User

Farm admin and Farmer.

Summary

To edit the equipment data a farm admin or farmer can go to the equipment view and there they are allowed to edit the data.

Priority

Should have

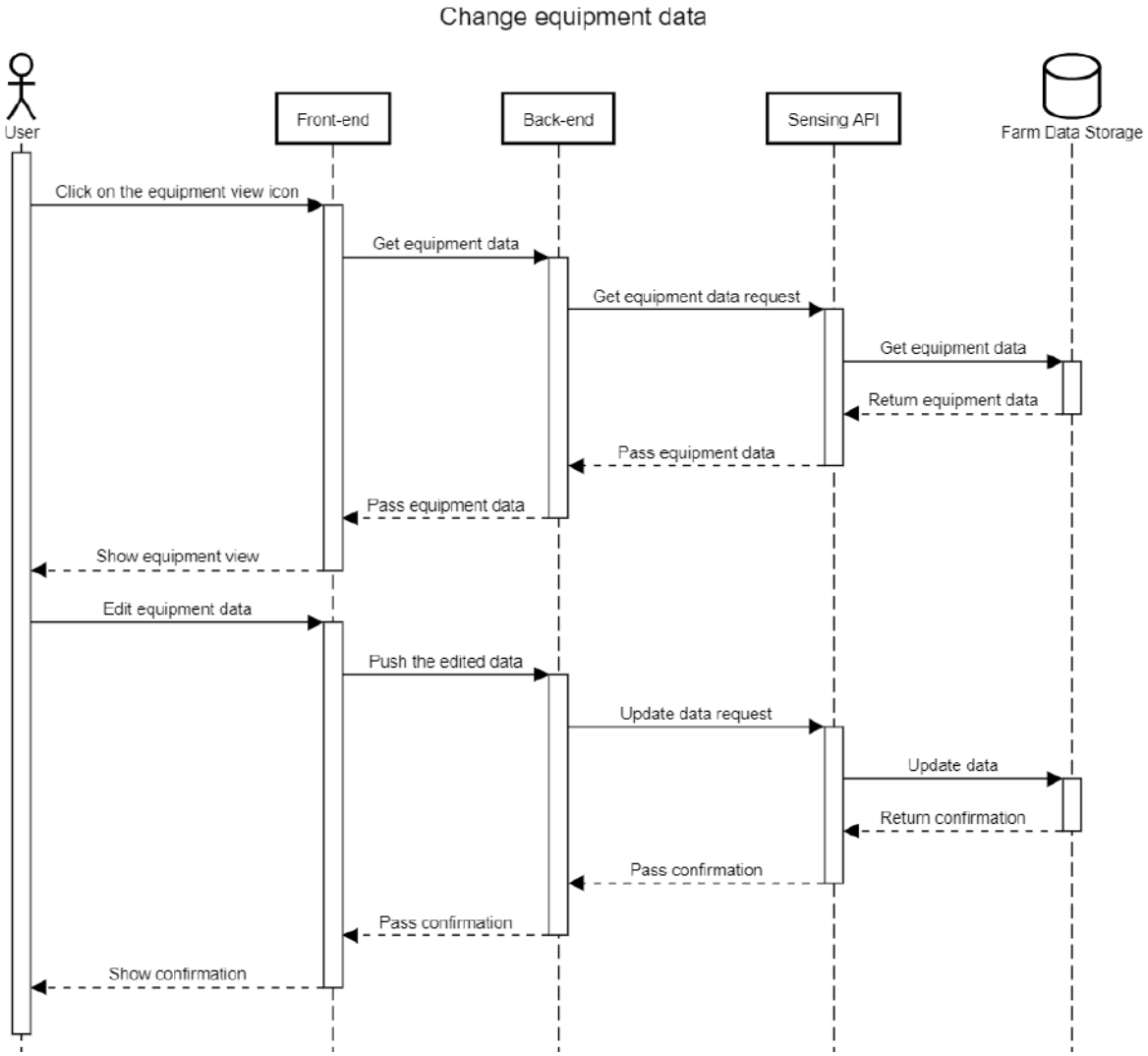

Figure 28: Change equipment data Sequence Diagram

3.3.21 Change observation data

Sensors can give wrong data if they are damaged. This means incorrect data can enter the database. To fix this, farm admins, farmers, and researchers can edit observation data.

Goal
To edit data that is on the history view.

Precondition
User is logged in and is a farm admin, farmer, or researcher. The users starts on the home view.

Postcondition
The data is edited and saved in the database.

User
Farm admin, Farmer, and Researcher.

Summary
The user goes to the history page and there edits the data so that the data is as the user wants.

Priority
Should have

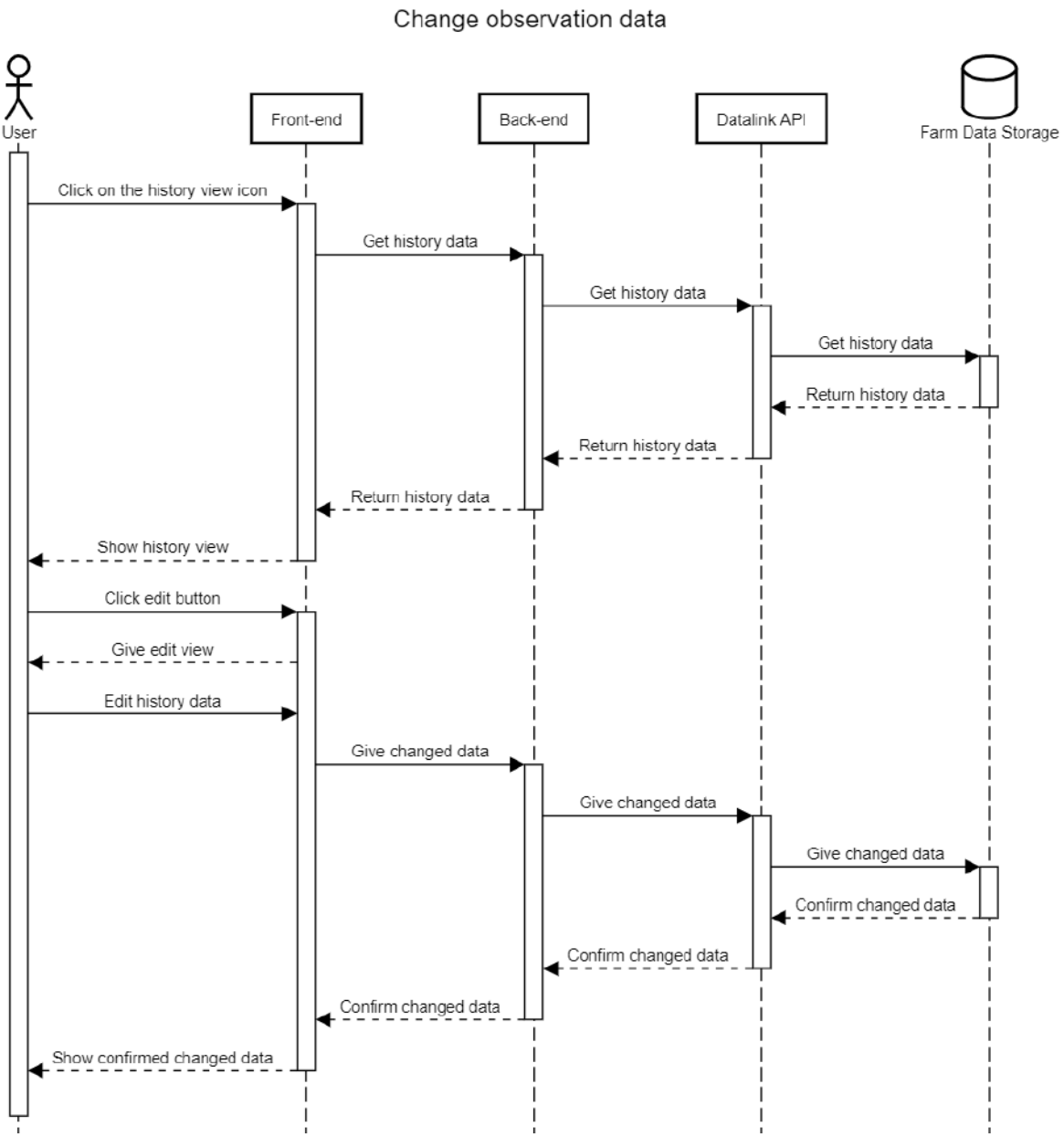

Figure 29: Change Observation data Sequence Diagram

3.4 Data model

This chapter provides an explanation of the data model used for the application, leading with a class diagram. This model is a high-level overview of the current system design and its corresponding sub-modules. The data model primarily displays which objects are linked to each other and how they are stored in the database. It is built of four modules; the user module (green), the permission module (yellow), the datamapping module (pink), and the farm data module (red).

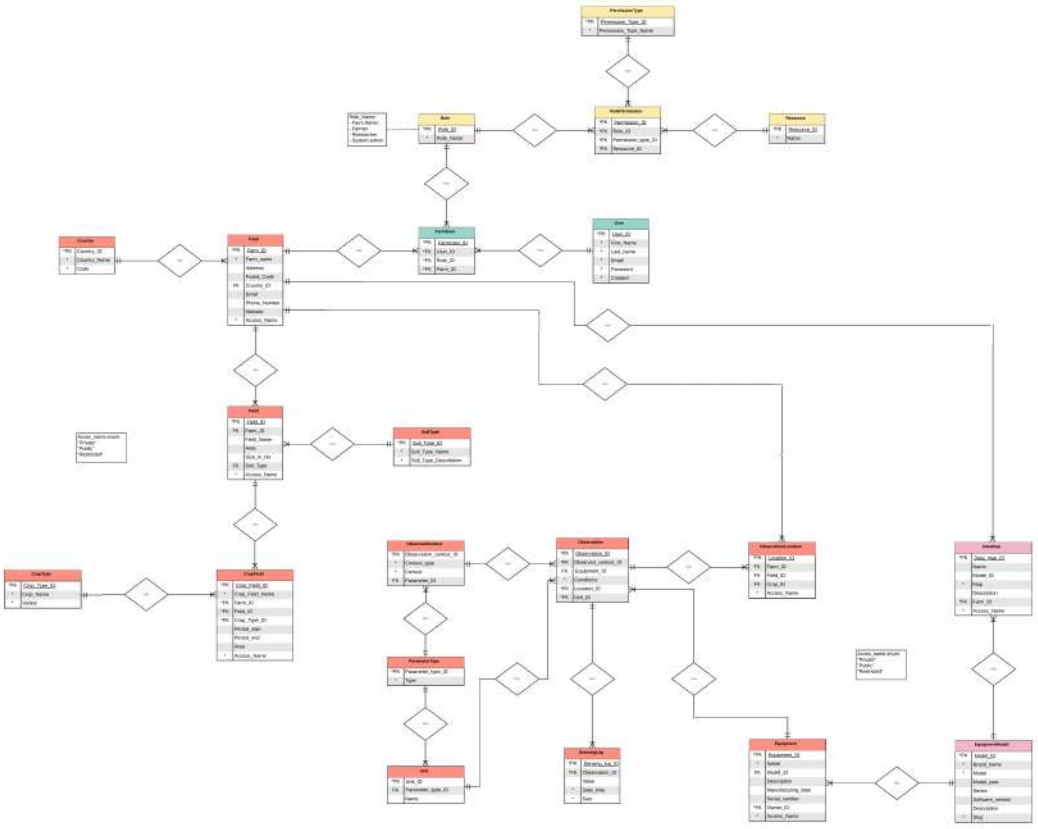

Figure 30: Data model

3.4.1 Permission module

PermissionType

A PermissionType object represents all possible permission types. To ensure modularity and allow for future expansion, this object is modelled separately.

- Permission_Type_ID: Each permission type is uniquely identified by its own ID, which is also the primary key for the object.
- Permission_Type_Name: The name of the permission type.

RolePermission

The RolePermission object stores a distinct ID value that defines the specific permissions of a role.

- RolePermission_ID: Each RolePermission is uniquely identified by its own ID, which is also the primary key for the object.
- Role_ID: Each Role is identified by its own ID, which is also the Foreign key for the object.
- Permission_Type_ID: Each permission type is identified by its own ID, which is also the Foreign key for the object.
- Resource_ID: Each Resource is uniquely identified by its own ID, which is also the Foreign key for the object.

Resource

A Resource object stores the type of object the permission should be allocated to.

- Resource_ID: Each Resource is uniquely identified by its own ID, which is also the primary key for the object.
- Resource_Name: A string containing the name of the Resource.

Role

The Role object details the name of the role and its distinct ID for the Permission.

- Role_ID: Each Role is uniquely identified by its own ID, which is also the primary key for the object.
- Role_Name: The name of the Role.

3.4.2 User module

FarmUser

A FarmUser object is a model containing a User object who has the Role object specific access rights in a farm.

- FarmUser_ID: Each FarmUser is uniquely identified by its own ID, which is also the primary key for the object.
- User_ID: Each User is uniquely identified by its own ID, which is also a Foreign key for this object.
- Role_ID: Each Role is uniquely identified by its own ID.
- Farm_ID: Each Farm is uniquely identified by its own ID, which is also a Foreign key for this object.

User

A User object contains the personal information of a user of the application. The user object has default permissions for all farms (none).

- User_ID: Each User is uniquely identified by its own ID, which is also the primary key for this object.
- First_Name: A string representing the first name of the User.
- Last_Name:A string representing the last name of the User.
- Email: A string representing the Email of the User.
- Password: A string representing the Password of the User.
- Created: A timestamp, which represents the time when the User is created.

3.4.3 Farm data module

Farm

The Farm object stores all basic information of a single farm, identified by a unique ID.

- Farm_ID: Each Farm is uniquely identified by its own ID, which is also the primary key for this object.
- Farm_Name: A string representing the name of the farm.
- Address: A string representing the address of the farm.
- Postal_Code: A string representing the postal code of the farm.
- Country_ID: Each Country is uniquely identified by its own ID, which is also the Foreign key for this object.
- Email: A string representing the Email of the farm.
- Phone_Number: A string representing the phone number of the farm.
- Website: A string representing the URL of the farm's website.

- Access_Name: An enum with the value Private,Public or Restricted.

Country

The Country object is contained in a list of Countries and has a unique ID, name, and code.

- Country_ID: Each Country is uniquely identified by its own ID, which is also the Primary key for this object.
- Country_Name: A string representing the name of the Country.
- Country_Code: A string which represents the country in two or three letters using the worldwide known ISO 3166-1 standard [**ISO 3166**].

Field

A Field object has a many-to-one relation with the Farm object and can have a single SoilType object.

- Field_ID: Each Field is identified by its own ID, which is also a Primary key for this object.
- Farm_ID: Each Farm is uniquely identified by its own ID, which is also the Primary key for this object.
- Field_Name: A string which represents the name of the field.
- Field_Area: A list of coordinates, which together form a polygon.
- Field_Size_In_Ha: A Double which defines the size of the field in hectares.
- Soil_Type_ID: Each Soil type is uniquely identified by its own ID, which is also a foreign key for this object.
- Access_Name: An enum with the value Private,Public or Restricted.

SoilType

The SoilType object specifies the corresponding type of soil for a farm field, including its unique ID, name, and description.

- Soil_Type_ID: Each Soil type is uniquely identified by its own ID, which is also a foreign key for this object.
- Soil_Type_Name: A string which represents the name of the soil type.
- Soil_Type_Description: A string which represents a short description about the soil type, light/heavy/warm/cold/dry/wet/low nutrients/high nutrients.

CropField

A CropField object has a distinct ID, Field ID, and Farm ID attached to it. It houses the period in which the cropfield is active and which CropType object is associated with it.

- Crop_Field_ID: Each crop field is identified by its own ID, which is also a Primary key for this object.
- Crop_Field_Name: A string which represents the name of the crop field.
- Farm_ID: Each farm is identified by its own ID, which is also a foreign key for this object.
- Field_ID: Each field is identified by its own ID, which is also a foreign key for this object.
- Crop_Type_ID: Each crop type is identified by its own ID, which is also a foreign key for this object.
- Period_Start: A timestamp, which represents the time when the crop will be planted.
- Period_End: A timestamp, which represents the time when the planted crop will be harvested.
- Crop_Field_Area: A list of coordinates, which together form a polygon.
- Access_Name: An enum with the value Private,Public or Restricted.

CropType

The CropType object specifies the corresponding type of crop for a cropfield, including its unique ID, name, and variety.

- Crop_Type_ID: Each crop Type is uniquely identified by its own ID, which is also a Primary key for this object.
- Crop_Type_Name: A string which represents the name of the crop.
- Crop_Type_Variety: A string which represents the variety of the crop.

ObservedContext

The ObservedContext object stores the type of observation made, its parameter, and its unit and attaches it to the Observation object.

- Observation_Context_ID: Each observation context is uniquely identified by its own ID, which is also a Primary Key for this object.
- Context_Type: A string representing the type of context for the observation (e.g. environment, crop).
- Context: A string which represents the context for the observation (e.g. soil).

- Parameter_ID: The id of the parameter linked with this observation context.

ParameterType

The ParameterType object is the parameter of the ObservedContext object and allows for easy repurpose for new observed contexts.

- Parameter_Type_ID: Each parameter type is uniquely identified by its own ID, which is also a Primary Key for this object.
- Type: A string representing the type of parameter (e.g. parameter).

Unit

The Unit object is part of a list of different units that an Observation object and ParameterType object can be divided in. Just as the ParameterType object, modeling the Unit object separately allows for quick reuse in different and/or new parts of the module.

- Unit_ID: Each Unit is uniquely identified by its own ID, which is also a Primary Key for this object.
- Parameter_Type_ID: A string representing the type of parameter (e.g. parameter).
- Name: A string representing the name of the unit.

Observation

The Observation object is the linking module for the ObservedContext, Unit, SensingLog, Equipment, and ObservationLocation objects. This object handles the combined data and location of a single observation on a specific spot in a farm.

- Observation_ID: Each Observation is uniquely identified by its own ID, which is also a Primary Key for this object.
- Observation_Context_ID: Each Observation is identified with an Observation Context ID, which is a Foreign key for this class.
- Equipment_ID: Each Observation is identified with an Equipment ID, which is a Foreign key for this class.
- Conditions: A JSON binding layer to convert the Java objects into JSON Messages.
- Location_ID: Each Location is identified by its own ID, which is also a Foreign Key for this.
- Unit_ID: Each Unit is identified by its own ID, which is also a Foreign Key for this object.

SensingLog

The SensingLog object stores the actual data and date of the observation and links it to the Observation object.

- Sensing_Log_ID: Each Sensing Log is uniquely identified by its own ID, which is also a Primary Key for this object.
- Observation_ID: Each Observation is identified by its own ID, which is also a Foreign Key for this object.
- Value: A double precision value of the measurement.
- Date_Time: A Timestamp which represents the time and date of the measurement.
- Geo: Coordinates which represents the location of the measurement.

Equipment

The Equipment object has a unique ID and contains the information of a sensor and its corresponding DataMap object to ensure the data is translated correctly in the system and is stores with data of the same type.

- Equipment_ID: Each Equipment is uniquely identified by its own ID, which is also a Primary Key for this object.
- Name: A string representing the name of the equipment.
- Model_ID: Each Model is identified by its own ID, which is also a Foreign Key for this object.
- Description: A string representing a description of the equipment.
- Manufacturing_Date: A time stamp representing the date the equipment is made.
- Serial_Number: A string representing the serial number of the equipment.
- Owner_ID: Each Owner is uniquely identified by its own ID, which is also a Foreign Key for this object.
- Access_Name: An enum with the value Private,Public or Restricted.

ObservationLocation

The ObservationLocation object connects the Observation object with the farm that it belongs to based on the location of the observation. It also stores the accessibility type of the observation, indicating who can see the observation.

- Location_ID: Each Location is uniquely identified by its own ID, which is also a Primary Key for this object.

- Farm_ID: Each Farm is identified by its own ID, which is also a Foreign Key for this object.
- Field_ID: Each Field is identified by its own ID, which is also a Foreign Key for this object.
- Crop_ID: Each Crop is identified by its own ID, which is also a Foreign Key for this object.
- Access_Name: An enum with the value Private,Public or Restricted.

3.4.4 Datamapping module

EquipmentModel

The EquipmentModel object links the Equipment object (sensor) to a DataMap object to ensure the data from the sensor is correctly converted in the database. It also contains some more details of the sensor.

- Model_ID: Each Model is uniquely identified by its own ID, which is also a Primary Key for this object.
- Brand_Name: A string representing the brand name of the equipment.
- Model: A string representing the name of the model.
- Model_Year: An integer representing the build year of the equipment.
- Series: A string representing the serie of the equipment
- Software_Version: A string representingt the Software version currently running on the equipment.
- Description: A string representing a descripting of the kind of equipment.
- Slug.

DataMap

A DataMap object contains the corresponding data columns in the map parameter, which indicate which columns of the incoming sensor or import data stores which type of observation data.

- Data_Map_ID: Each Data Map is uniquely identified by its own ID, which is also a Primary Key for this object.
- Name: A string representing the name of the Data Map.
- Model_ID: Each Model is identified by its own ID.
- Map: A JSON binding layer to convert the Java objects into JSON Messages.

- Description: A string representing a description of the Data Map
- Farm_ID: Each Farm is uniquely identified by its own ID, which is also a Foreign Key for this object.
- Access_Name: An enum with the value Private,Public or Restricted.

3.5 External interface definitions

This section will give a description of the external systems that CloudFarmer interacts with as well as the interfaces it uses to make sure the external systems work.

3.5.1 Dacom's API

CloudFarmer uses Dacom's API to retrieve sensors' measurements from Dacom's database. These measurements are crucial for CloudFarmer in its early stages. In order to extract these measurements, the backend sends the following request to the database: GET/API/ v3/ farms/ <farm_id>/installations/<installation_id>/interpreted_measurements/. Consequently, this returns a list of interpreted measurements that is based on the settings made by the user.

3.5.2 WolkyTolky API

CloudFarmer uses the WolkyTalky API to access an overview of sensors as well as their respective measurements. This API supports reading of data only.

The WolkyTolky API supports the following functionalities:

- Retrieving an overview of the sensors on a farm: For users to have access to the overview of the sensors used in a farm, they have to enter their API key which WolkyTolky provided. If the API key is correct, then the result is returned by default as a CVS file. However, the user can opt for the file extension of the result to be JSON, XML or as an Excel file.
- Retrieving sensors' measurements between two specific dates at most 1 year apart: To acquire the measurements of a sensor in a certain period, the user has to provide the sensor id, start date and end date. Furthermore, the user has to provide the correct API key. The user is free to choose the file extension of the result; either a CSV, JSON, XML or an Excel file.
- Retrieving sensors' locations on a farm: To acquire the GPS location of a sensor, the user has to enter the sensor ID accompanied by the correct API key. Similar to the previous two functions, the default format of the result is CSV, however, the user can alter it to JSON, XML or Excel.

3.5.3 Data Mapping API

The Data Mapping API is responsible for creating and controlling all the datamaps that are needed to store the collected data in the right way. If data has the same datamap it can be combined together into graphs and tables.

- Retrieving a list of data mapping by sending the request `GET /datamaps/` to the server.
- Adding a new data map by sending the request `POST /datamaps/` to the server.
- Returning a data map by ID by sending the request `GET /datamaps/{map_id}` to the server.
- Updating an existing data map by ID by sending the request `POST /datamaps/{map_id}` to the server.
- Deleting an existing data map by ID by sending the request `DELETE /datamaps/{map_id}` to the server.
- Returning a list of equipment models by sending the request `GET /datamaps/models` to the server.
- Adding a new equipment model by sending the request `POST /datamaps/models` to the server.
- Returning an equipment model by slug by sending the request `GET /datamaps/models/{slug}` to the server.
- Deleting an equipment model by slug by sending the request `DELETE /datamaps/models/{slug}` to the server.

3.5.4 Sensing API

The Sensing API is responsible for getting and storing most of the data on the Farm Data Storage database. This way all the information about the farms, fields, crop fields, equipment and observations will be saved.

- Retrieving a list of farms by sending the request `GET /farms` to the server.
- Adding a new farm by sending the request `POST /farms` to the server.
- Returning farm information by ID by sending the request `GET /farms/{farm_id}` to the server.
- Updating farm information by ID by sending the request `PUT /farms/{farm_id}` to the server.

- Deleting a farm by ID by sending the request
  `DELETE /farms/{farm_id}` to the server.
- Returning a list of fields in a farm by sending the request
  `GET /farms/{farm_id}/fields` to the server.
- Adding a new field to a farm by sending the request
  `POST /farms/{farm_id}/fields` to the server.
- Returning field information by ID by sending the request
  `GET /farms/{farm_id}/fields/{field_id}` to the server.
- Updating field information by ID by sending the request
  `PUT /farms/{farm_id}/fields/{field_id}` to the server.
- Deleting a field by ID by sending the request
  `DELETE /farms/{farm_id}/fields/{field_id}` to the server.
- Returning a list of crop fields in a field by sending the request
  `GET /farms/{farm_id}/fields/{field_id}/crop_fields` to the server.
- Adding a new crop field to a field by sending the request
  `POST /farms/{farm_id}/fields/{field_id}/crop_fields` to the server.
- Returning crop field information by ID by sending the request
  `GET /farms/{farm_id}/fields/{field_id}/crop_fields/{crop_field_id}` to the server.
- Updating crop field information by ID by sending the request
  `PUT /farms/{farm_id}/fields/{field_id}/crop_fields/{crop_field_id}` to the server.
- Deleting a crop field by ID by sending the request
  `DELETE /farms/{farm_id}/fields/{field_id}/crop_fields/{crop_field_id}` to the server.
- Retrieving a list of crop types by sending the request
  `GET /crop_types` to the server.
- Adding crop type by sending the request
  `POST /crop_types` to the server.
- Retrieving equipment information by ID by sending the request
  `GET /equipments/{equipment_id}` to the server.
- Updating equipment information by ID by sending the request
  `POST /equipments/{equipment_id}` to the server.
- Deleting equipment information by ID by sending the request
  `DELETE /equipments/{equipment_id}` to the server
- Retrieving a list of the farms' countries by sending the request
  `GET /country_list` to the server.
- Retrieving a list of soil types by sending the request
  `GET /soil_types` to the server.
- Retrieving observation data by sending the request
  `GET /observations` to the server.

- Importing observation data by sending the request `POST /observations/upload` to the server.

3.5.5 User Auth API

The User Auth API is responsible for the authentication and the authorisation of the users. This means it will check the user credentials if they want to login. It will save the user credentials if they create an account. It will also save the roles the user has on each farm to make sure the user can only see and edit data that they are allowed to see.

- Returning a list of users by sending the request `GET /users` to the server.
- Creating a new user by sending the request `POST /users/register` to the server.
- Returning user information by ID by sending the request `GET /users/<user_id>` to the server.
- Updating user information by ID by sending the request `PUT /users/<user_id>` to the server.
- Deleting a user information by ID by sending the request `DELETE /users/<user_id>` to the server.
- Returning a list of user roles by sending the request `GET /roles` to the server.
- Creating a new user role by sending the request `POST /roles` to the server.
- Returning a user role by ID by sending the request `GET /roles/<role_id>` to the server.
- Updating a user role by ID by sending the request `PUT /roles/<role_id>` to the server.
- Deleting a user role by ID by sending the request `DELETE /roles/<role_id>` to the server.
- Returning a list of role permissions by sending the request `GET /roles/permissions` to the server.
- Creating a new role permission by sending the request `POST /roles/permissions` to the server.
- Returning a list of role permissions by ID `GET /roles/permissions/<role_id>` to the server.
- Updating role permissions by ID by sending the request `PUT /roles/permissions/<role_id>` to the server.

- Deleting a role permission by ID by sending the request
  DELETE /roles/permissions/<role_id> to the server.
- Returning all users who have role access rights in a particular farm by sending the request
  GET /farm_users/<farm_id> to the server.
- Creating new role access rights of an user in a particular farm by sending the request
  POST /farm_users/<farm_id> to the server.
- Returning role access rights of a user in a farm by sending the request
  GET /farm_users/<farm_id>/users/<user_id> to the server.
- Updating role access rights of a user in a farm by sending the request
  PUT /farm_users/<farm_id>/users/<user_id> to the server.
- Deleting role access rights of a user in a farm by sending the request
  DELETE /farm_users/<farm_id>/users/<user_id> to the server.
- Returning all user roles in farms by sending the request
  GET /farm_users/<user_roles>/<user_id> to the server.
- Returning JWT tokens for authorization by sending the request
  POST /auth/jwt_token to the server.
- Returning new access JWT tokens by sending the request
  GET /auth/refresh_token to the server.
- Verification of resource access permission by sending the request
  POST /auth/access_verification to the server.

3.5.6 DataLink API

The DataLink API is used to get data from the WolkyTolky AP, the Dacom API and the Farm Data Storage. This API is used for the live view and the history view.

- Returns a list of WolkyTolky equipment by sending request GET /equipments_wt/ to the server.
- Returns WolkyTolky equipment by its ID by sending request GET /equipment_wt/ to the server.
- Register a new WolkyTolky equipment with API key by sending POST /equipment_wt/ to the server.
- Update WolkyTolky equipment by its ID by sending request PUT /equipment_wt/ to the server.
- Delete WolkyTolky equipment by its id bly sending request DELETE /equipment_wt/ to the server.

- Returns a list of the latests observation data from all sensor in the specified area by sending request GET /live/ to the server.

3.6 Design rationale

In this section, the design decisions are explained and justified using comparisons of multiple technologies as well as the use case and scope of the application.

3.6.1 Separation of Front-end and Back-end

The applications architecture has a clear distinction between the front-end and the back-end. The front-end arranges how information is displayed to the client and how the client interacts with the application. The back-end handles how the front-end communicates with the server and the micro-services. However, both the front-end and the back-end are run client-side, while the server acts as a router for incoming and outgoing requests from and to the database. This allows for tighter integration of the back-end in the front-end functionality and thus facilitates easier communication between both platforms.

The front-end and back-end both use Typescript as the programming language and React as a universal framework. By using both the same language and framework for our application, we fade the border between front-end and back-end to allow for better integration and user interactivity. However, the back-end functions are still separated from the front-end functions, but still easily changed by anyone as we use a single language.

3.6.2 Front-end Language

The front-end portion of CloudFarmer has been most primarily built using Typescript with some components written in JavaScript to ensure compatibility with some libraries.

The primary advantages of using JavaScript is the immense support base and the creation of interactive web pages. As TypeScript is a superset of JavaScript, it allows us to use the existing functionality of Javascript with the added robustness of Typescript. The rationale for using Typescript is to prevent errors related to type definitions since TypeScript supports static typing. [25]

On the deployment of the application, the TypeScript code is compiled to JavaScript code, so that the code is compatible with the large expanse of web browsers. As TypeScript is built upon the functionality of JavaScript, there also exists a vast range of available documentation and support, which improves the speed of bug discovery and fixing.

The only other language that is supported in modern browsers is WebAssembly, but this is yet to be as widely supported as JavaScript on all browsers, requires knowledge of C++, and does not offer the same level of interactivity. [28] However, there are still reasonable alternatives for TypeScript, such as:

Dart

Dart is a language created by Google that compiles to JavaScript. [7] It supports static typing, which is more flexible than TypeScript's typing. However, Dart can also be used for non-web applications and is therefore not well supported by React. It also does not integrate well with other JavaScript libraries for the same reason, which would complicate the development of the application.

Flow

Flow is a static type checker for JavaScript. [10] Similar to TypeScript, it needs a compiler to check the types and remove the types from the JavaScript code, as JavaScript does not allow types by default. However, it is less widely used than TypeScript and there would thus be less support for types in third-party packages that are required for the development.

Kotlin

Kotlin is a language that was originally made for the Java VirtualMachine, therefore making it very similar to Java, including full static typing. [14] It can compile down to JavaScript, however it has limited support for React and it does not integrate well with JavaScript libraries. Even more, the documentation for Kotlin and especially Kotlin.js is way too sparse to be used in an open-sources application.

3.6.3 Front-end Framework

The front-end framework of the application is based on React, a library based on JavaScript. [23] React is mainly chosen because it facilitates building modular applications. This allows the reuse of UI components as well as dynamically changing their data.

Consequently, using React makes it easier to define and manipulate these components and thus simplifying code maintenance and growth. As the application will mostly be used as a single-page application, React fulfils this niche quite well. React makes it simple to define modular components based on states set by the user. This is why we have a tighter integration between the front-end and the back-end, as we specify most functions on a component basis, instead of pure render and data handling functions.

React allows for greater flexibility in choosing which libraries and functions are used, which reduces the import footprint. React also utilises a virtual DOM, which only updates changed components instead of refreshing the entire tree structure up to the change. React also has a large ecosystem combined with a vast support base. Lastly, the directory and file structure has a clear and simple layout, which makes it easy to view test cases per file and to locate assets.

Angular was also considered to be used as the framework for the application's front-end. [5] However, Angular was discarded since React offers a one-way binding of data, whereas Angular's data binding is two-way. In other words, if a UI element is changed in React, the model state does not automatically change, as opposed to Angular where it does change. The latter contributes to code that is easier to understand, but it does not scale well for large projects and eventually degrades the debugging experience. Angular

also uses a regular DOM, instead of a virtual DOM, which greatly increases the time needed to compile new changes to the application.

Vue is also a framework that was considered to be used for the application. [27] It offers a lightweight, but still sophisticated core with many default functionalities. Vue has, like React, a virtual DOM, for faster compiling during the development process. Vue is a relatively new framework and builds upon the older frameworks and thus combines the pros from these frameworks. Being a newer piece of software comes with the downside of a smaller community and thus less support for problems that might occur during the development. As Vue felt a bit too experimental, we opted not to use it for the application.

3.6.3.1 Material UI

For the general layout of our user interface, we decided to use the Material UI framework that creates a very nice looking default theme that can be easily adapted to fit our needs. [16] With JSS fully integrated into Material UI, we can add dynamic CSS to insert in our existing HTML. As Material UI is based on the Material Design by Google, its style allows for an uniform, flexible application that creates a clear overview for our users.

3.6.4 Back-end Language

To create a unified work environment, we decided to use Typescript (explained above) as our primary programming language for the back-end. By using Typescript for the front-end as well as the back-end, all application syntax is written in a single language. Typescript is extremely useful for the back-end as it streamlines the process of creating and maintaining Promises. Promises are a method for creating asynchronous tasks that can be used to retrieve data from a database. Promises also allow for near-infinite chaining, which is necessary to facilitate easy-to-understand production processes.

Python was also an option to use for our back-end systems but failed to comply with the standards of a large scale application. [22] Python, on one hand, has an immense ecosystem and a massive list of libraries, which are mostly used in sciences and machine learning. On the other hand, Python is very error-prone and hard to maintain due to it being indentation dependent. Python also performs much slower on basic tasks, which makes it unsuitable for an interactive single-page application.

Another option was to use the old, but still a quite popular language PHP. [20] PHP is very easy to implement into already existing web pages due to its direct integration into the HTML. It combines easily with several different database types, like MySQL and PostgreSQL. However, as we have to deal with a lot of JSON strings that are sent by the already existing API, PHP is harder to use due to it being less supported than its database types. PHP also has a very steep learning curve, which is not the most optimal for beginner web developers like us.

3.6.5 Back-end Framework

We mostly based our decision for the back-end on the frameworks available and their complicity. To sufficiently support the back-end language, we employ NodeJS as runtime environment for our asynchronous background processes. NodeJS is based on the V8 JavaScript engine developed by Google and used for Chromium browsers.[19] This means that NodeJS is extremely fast and optimised by experienced developers.

The main functionality of NodeJS is that it allows the process to continue operations while the server works on the sent request. When the server is done, NodeJS automatically retrieves the data, which can then be used without halting the requests, unlike with PHP and ASP. This allows the application to handle server requests in real-time and creates a more interactive experience for the user.

Other back-end frameworks for other languages, which we do not use, are:

Laravel
: Laravel is a back-end framework for PHP, which is very popular among PHP developers. [15] Laravel is designed based on the Model-View-Controller architecture. It has a very strong module bundler and unit testing system. As Laravel handles HTTP requests and JSON strings less elaborate than NodeJS, we opted not to choose PHP and in turn Laravel.

Ruby on Rails
: Ruby on Rails is a fully integrated server-side Model-View-Controller framework that written in Ruby. [24] As it is fully integrated, everything mostly already works out of the box. Rails has an extensive infrastructure that makes it easy to set up a base for the application. A problem with this is the decreased flexibility in the modules and the massive amount of already imported modules, which can slow down the loading times. As none of the group members has used Ruby before, we decided to skip it to avoid having to spend too much time learning Ruby, while it may not be as profitable as using another language.

Django
: Django is a Python based framework that brings a lot of features out of the box. Django is very beginner friendly due to its inclusive packaging and Python being its default language. Django is extremely scalable, more so than NodeJS, but it performs worse in interactive single-page applications. Its performance deficit is mostly deceived from its server processes, which cannot be handled asynchronously. To utilise Django to its greatest potential, the Model Template View architecture has to be clearly understood, which is something none of our members had experience with.

3.6.5.1 Express

Express is a NodeJS web application framework that facilitates the dynamic loading of views based on content retrieved from the server and defines a routing table that can be used to direct HTTP requests. [8] It handles the asynchronous methodology from

NodeJS by itself, thus simplifying the implementation. Express is used to create middleware for our NodeJS requests, which allows us to convert data types using a predefined template. It also lets us define generic error handling on the back-end, so that the front-end can send generalised feedback when a server error occurs.

3.6.5.2 PostgreSQL

The application has to store some personal user information and all of the farm and observation data. To facilitate this, we decided to use two centralised databases that the API can communicate with. The databases are managed by PostgreSQL on request by the client. [21] The client has already provided the databases and parts of the microservices, so it would be sensible to use this system for our application as a whole.

3.6.5.3 Jest

Jest is a JavaScript test runner, which provides the foundation of unit tests for the application's back-end and part of the front-end. [13] Using Jest we can create assertions so that each test case is custom tailored to our implementation. Jest is also extremely fast as it runs all tests in parallel, which is a necessity with such a large application. By creating dummy data, also called *mocking* [18], parts of the application can be simulated in a test environment to ensure the eventual data will be handled correctly. Jest also has default checking functions, called *matchers* that allows for simple testing functions.

3.6.6 Continuous Integration and Deployment

To allow for smooth development and concise, consistent code, we set up a continuously integrated development environment. For this, we use Gitlab to set up a centralised DevOps lifecycle for our development.[11] Gitlab grants us the option to use the functionality of Git together with specific testing and deployment pipelines for each branch. These pipelines also run all tests again (in combination with local tests) to ensure that merging branches do not break the deployed code.

As we only have a single language for both the front-end and the back-end we do not need to use multiple different testing libraries and we can fully test our application with Jest.

4 Feasibility and Resource Estimates

This section gives an estimate of the hardware necessary to run the development, client, and server machine for the application. It also provides a performance metric based on the listed resource requirements and the performance requirements listed in the URD.

4.1 Resource Requirements

4.1.1 Development machine minimum requirements

CPU Any capable of running the operating systems below

Operating System Microsoft Windows, Mac OS or Linux

Memory Minimum 1GB

Disk Space Minimum 2GB

Network At least 1 Mbit/s for responsive usage (none needed for local testing)

Browser Chromium $\geq$ 72, Safari $\geq$ 12, Firefox $\geq$ 64,

Other Software PostgreSQL $\geq$ 11

4.1.2 Client machine requirements

CPU Any capable of running the operating systems below

Operating System Any capable of running a browser from the ones listed below

Memory Minimum 1GB

Disk Space 1MB for cookies and local cache

Network At least 1 Mbit/s for responsive usage

Browser Chromium $\geq$ 72, Safari $\geq$ 12, Firefox $\geq$ 64,

Other Software PostgreSQL $\geq$ 11

4.1.3 Server machine requirements

CPU Any capable of running the operating systems below

Operating System Microsoft Windows, Mac OS or Linux

Memory Minimum 2GB

Disk Space Minimum 4GB

Network At least 1 Mbit/s for every connection to the application's server

4.2 Performance

4.2.1 Measuring performance

The performance of the website can be tested using Chromium's integrated DevTools performance metrics. These tools create default measurements according to Google's standard testing procedure based on the RAIL model. [17] As most standard other tools could not optimally take measurements on a single page application, we opted to use Google's own integrated tools.

4.2.2 Test setup

The performance testing was done on a member's personal desktop. The website was run locally on Chrome Version 78 (64-bit). The OS of the testing machine is Windows 10 Pro Build 1809. The hardware specifications are as follows:

CPU 3.4GHz 6 cores.

Memory 16GB.

Network 1 GBit/s, throttled to 100 Mbit/s for average use.

The tests are done on a test account and a standard farm is selected to conduct the default load times for the farm's components.

4.2.3 Result explanation

The results of the performance testing can be separated into different events, specified by the User-centric Performance Metrics page. [26] These include:

DOMContentLoaded This event is fired when all of the page's DOM content has been loaded and parsed.

First Paint This event marks the point when the browser renders anything that is visually different from what was on the screen prior to navigation.

First Contentful Paint This event marks the point when the browser renders the first bit of content from the DOM.

Onload Event This event marks the point where the DOM can fire additional application logic.

Largest Contentful Paint This event marks when the largest element on the page has loaded.

First Meaningful Paint This event is the metric shows the most important part of the page (hero elements).

As the website is a single page application, the first few columns are the same for each view. The time it takes to load a specific view is measured by the difference of the Largest Contentful Paint and the First Meaningful Paint. This is only not true for the Login page, as that page does not have separate views that have to be loaded, meaning the difference will be zero. Other metrics are also useful for performance, like the total size of the requested data and the amount of requests on page load.

4.2.4 Performance results

DOMContentLoaded, First Paint, First Contentful Paint, Onload Event, Largest Contentful Paint, First Meaningful Paint are all measured in milliseconds.

DOM Content Loaded	First Paint /ms	First Contentful Paint /ms	Onload Event /ms	Largest Contentful Paint /ms	First Meaningful Paint /ms
1075.8	1103.6	1103.6	1285.4	1296.6	2040.4

As seen in the table above, the average loading time of the application is about 2 seconds, which is normal for most single page applications in 2019. [12]

Largest Contentful Paint, First Meaningful Paint, and Time between paints are measured in milliseconds. The Resources column is measured in Megabytes and the Requests column is measured in total requests on page load.

Page	Largest Contentful Paint /ms	First Meaningful Paint /ms	Time between paints/ ms	Resources /Mb	Requests
Login	1937.6	1937.6	0	22.7	16
Farms	1140.6	1941.3	800.7	10.7	20
Live	1214.2	1722.9	508.7	10.6	38
History	1143.1	1543.4	400.3	10.6	20
Fields	1218.1	1968.7	750.6	11.9	125
Datamaps	1171.1	1988.5	817.4	10.8	22
Equipment	1234.6	2622.6	1388.0	10.8	32
Personal	1313.5	2597.9	1284.4	10.8	37
Average	1296.6	2040.4	743.7625	12.4	38.8

The Fields view performance metric in the table only accounts for the time loading the view, but not the Google Maps component. This component took until 2504.6ms to load, setting the total time for the view to load to 1286.5ms.