Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Jonas Ege Carlsen

Digitalizing the Sheep Supervision Documentation Process

Master's thesis in Informatics
Supervisor: Svein-Olaf Hvasshovd
June 2021

NTNU
Norwegian University of
Science and Technology

Jonas Ege Carlsen

Digitalizing the Sheep Supervision Documentation Process

Master's thesis in Informatics
Supervisor: Svein-Olaf Hvasshovd
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

A Norwegian sheep spends large parts of the grazing season on outlying fields. Farmers are mandated by law to conduct weekly supervision trips to ensure the safety and well-being of their animals, a process currently documented using pen and paper. Once the grazing season ends, supervision-related documentation is summarized into a seasonal report and sent to governing bodies as proof of sufficient supervision and as a basis for potential reimbursement claims. This thesis aimed to develop a digital system replacing the analog supervision documentation process, and explore the potential benefits such a solution could provide to farmers, sheep observers and the well-being of sheep. The project resulted in a cross-platform mobile application for conducting supervision trips, a server for centrally storing supervision data and a surrounding cloud infrastructure supporting it.

The ongoing COVID-19 pandemic rendered attempts at usability testing farmers and sheep observers futile. Instead, the usefulness of the system had to be discovered through other means. By utilizing usability testing, prototypes and discussions with the project supervisor, a seasoned sheep observer, the project uncovered several potential benefits the digitalized documentation process could introduce. Usage of the system can improve the precision, structure and quantity of collected data, indirectly affecting the well-being of sheep by allowing farmers and observers to gain a greater overview of the whereabouts of predators and sheep alike. Furthermore, digital storage and the concept of supervision teams can facilitate communications between farmers and observers and optimize supervision routes, improving the overall efficiency of supervision.

Additional benefits can be extracted from the system by developing a web application tailored towards farmers, third-party integrations and functionality for automatically generating supervision reports. In order to facilitate such extensions, the system has focused on modifiability, extensibility and documentation. In large, the resulting system serves as a solid and extendable foundation of data creation and aggregation.

Sammendrag

Norsk sau tilbringer store deler av beitesesongen i utmark. Sauebønder er lovpålagt å føre tilsyn av sau minst en gang i uken for å passe på sikkerheten og velværen til dyrene deres, en prosess som til dags dato dokumenteres med penn og papir. Når beitesesongen avsluttes samles all tilsynsdata i en sesongrapport som fungerer som bevis for tilstrekkelig tilsyn, og som et grunnlag for potensiell kompensasjon for tapt sau. Denne oppgaven hadde som mål å utvikle et digitalt system for å erstatte den analoge løsningen for dokumentering av tilsyn, samt å utforske de potensielle fordelene en slik løsning kunne medføre for sauebønder, tilsynsmenn og sauens velvære. Prosjektet resulterte i en kryss-plattform mobilapplikasjon til bruk under tilsynsturer, en server for sentral lagring av tilsynsdata og en sky-infrastruktur som støtter systemet i sin helhet.

Den pågående COVID-19-pandemien førte til at brukertester ikke kunne gjennomføres på bønder og tilsynsmenn; Prosjektet måtte derfor finne resultater på andre måter. Gjennom brukertester, prototyper og samtaler med veileder, en erfaren tilsynsmann, fant prosjektet flere mulige fordeler en kunne oppnå ved å digitalisere tilsynsprosessen. Bruk av systemet kan forbedre presisjonen, strukturen og kvaliteten av den samlede data, noe som indirekte vil påvirke sauens velvære ved å tillate bønder og tilsynsmenn å få en bedre oversikt over hvor sauer og rovdyr befinner seg. Videre vil digital lagring av tilsynsdata og digitaliseringen av beitelag fasilitere kommunikasjon mellom forskjellige ansatte på tvers av gårder og gjøre det mulig å optimere tilsynsruter, noe som vil forbedre effektiviteten av tilsynsprosessen.

Videre fordeler kan ekstraheres fra systemet ved å utvikle en nettside rettet mot sauebønder, tredjeparts-integrasjoner og funksjonalitet for automatisk generering av tilsynsrapporter. For å legge til rette for slike utvidelser har systemet fokusert på modifiserbarhet, utvidbarhet og dokumentasjon. Systemet fungerer i stor grad som en solid og utvidbar plattform for generering og aggregering av tilsynsdata.

Acknowledgements

The author would like to express his gratitude to the project supervisor, Svein-Olaf Hvasshovd, for his continuous guidance and support. Your domain expertise and feedback proved to be invaluable.

Contents

Figures

Tables

Code Listings

Acronyms

API Application Programming Interface. 35, 48, 51, 57, 58, 87, 88, 94, 98, 104, 113, 114, 121, 122, 125, 136, 137, 148, 170, 175, 176, 178, 182
AWS Amazon Web Services. xvii, 37, 46, 136, 157, 162, 163, 168–183

CI/CD Continuous Integration / Continuous Deployment. 173
CLI Command Line Interface. 142, 179, 182

DAO Data Access Object. xiii, xv, 93, 96, 98, 107–111, 118, 119, 127, 139, 144
DTO Data Transfer Object. xiii, 138, 141, 142, 145, 148, 149, 155, 156, 159

GCP Google Cloud Platform. 104, 168

IDE Integrated Development Environment. 48, 58–60
IETF Internet Engineering Task Force. 87, 112, 175
IoC Inversion of Control. xv, 91, 92, 116, 117, 139, 147

JWT JSON Web Token. 48, 137, 145, 151

MVVM Model-View-ViewModel. xii, 91, 92, 101, 127

NSG Norsk Sau og Geit. xi, 7, 13, 128

ORM Object-Relational-Mapper. xv, 136, 139, 141–143, 145, 152, 154, 155, 164, 165

RDBMS Relational Database Management System. 133, 134, 158

SDLC Software Development Life-Cycle. 15, 16, 41, 42, 47, 125
SNO Statens Naturoppsyn. 5, 8, 9, 191
SQL Structured Query Language. 48, 103, 133, 164, 165

VPC Virtual Private Cloud. 170, 172
VSCode Visual Studio Code. 48, 60

WIP Work In Progress. 42, 44

Glossary

CRUD The four functionalities required in order to store data: Create, Read, Update and Delete. 109, 138, 139

EC2 A resizable computing service on AWS. 170

IaaS Providing and maintaining a set of virtualized computer resources for a premium. 168

IaC The process of creating infrastructure through the use of cloud-provider specific code, as opposed to utilizing user interfaces. 168, 176, 177

PaaS Providing a fully maintained development environment on a virtual computer resource. 168

SaaS A fully provided and maintained system that can be configured to fit specific customer needs. 38, 47, 168

YAGNI An acronym for "You ain't gonna need it!". Used to promote the practice of only implementing functionality when it is required. 109, 113

Part I

Project

Chapter 1

Introduction

Tending to domesticated animals is a practice as old as Norway itself, with farms spread throughout the country. Allowing domesticated animals to graze on outlying fields is considered a tradition and, at times, a necessity. By allowing their animals to graze in outlying fields, farmers are able to save money on food, while also ensuring that the animals are active and content.

The grazing process is different for every kind of domesticated animal. Norwegian sheep are left to graze on outlying fields throughout the entire grazing season. While this is beneficial for both farmers and sheep, it is not without its own problems. Throughout the 2019 grazing season, 1 933 947 sheep and lambs were released, of which 102 022 never returned [1]. Losses can be attributed to several factors: illness, separation from the herd, accidents and predators.

Of the aforementioned causes, deaths caused by predators are of special interest to farmers. Sheep predators are protected by law, and can therefore not simply be killed. Due to this, the Norwegian government offers reimbursements to all farmers who have suffered a loss of sheep due to predators. Rovbase, a collaborative tool for predator monitoring, reported a total of 33,605 sheep deaths that could be directly attributed to predators in 2019. Of these deaths, a total of 17,569 were reimbursed by the government [2].

In order to document sheep counts throughout the season, and to ensure their well-being, farmers have employees who conduct trips in known outlying fields to observe sheep throughout the season. These employees will be referred to as *sheep observers* throughout the rest of the thesis. The Norwegian government mandates that supervision trips are to be conducted at least once a week [3]. The observer notes down each and every sheep observation, regardless of whether the sheep are alive or not. Once the season is over, the observations are converted to a report, which in turn is used as the basis for any government reimbursements.

Currently, observations are neither standardized nor mandated; The government simply requires farmers to perform them in order to prove the well-being of grazing sheep and be eligible for reimbursements. Each farm has their own way of performing observations, leading to differing degrees of documentation. By standardizing observation documentation, the actors associated with the process would be able to utilize the generated data to a greater extent. It could allow for farmers to gain a greater overview of where their sheep are at, where predators have been spotted, or troubling areas where sheep have disappeared. A digital solution for documenting sheep observations could possibly make supervision trips more effective; Efficient user interfaces could increase the speed of which observations can be documented. Furthermore, a standardized and digital solution could automate processes such as reporting dead sheep or predators to the appropriate organizations. By having a standardized and thorough way of documenting dead sheep, farmers could have a better chance of being reimbursed for sheep deaths caused by predators. Finally, the end-of-season report required by the government can be automatically generated, significantly reducing the chance of human error.

This thesis will describe the development and usage of a system allowing sheep observers to digitally document their supervision trips, and share them with other sheep observers. It will describe the underlying development process, discuss how and why solutions were chosen, and how the system can be extended for future use. In doing this, the author aims to answer the following research questions:

- **RQ1:** What potential benefits can sheep observers and farmers gain by utilizing digital supervision solutions?
- **RQ2:** What is to be gained from collecting structured sheep supervision data?
- **RQ3**: In what ways can digitalized sheep supervision affect the well-being of grazing sheep?

1.1 Project Description

The aim of this project is to develop a system that aids sheep farmers and observers throughout the sheep grazing period. By introducing a cross-platform mobile application, sheep observers will be able to document their trips in a standardized manner through the use of a smart phone. The application will be fully capable of offline usage, to allow observers to benefit from it regardless of internet connectivity.

In order to be beneficial to sheep observers, the application must support the documentation of several key factors of a supervision trip. The user location will be continuously tracked in order to enable analysis at a later point. Furthermore, it will allow sheep observers to document several types of observations. Sheep observations will require the observer to document the number of sheep, their

whereabouts and their wool colors. They can optionally also document their tie colors and what farms the herd belongs to. The application should also support other types of observations, such as dead sheep and predators.

To allow for an effective method of conducting supervision trips, the system will support the concept of supervision teams: A group of sheep observers working together in teams, allowing for a greater overview of all sheep grazing within an area.

The system will include a cloud-based server, to allow for centralized storage of the data generated by the mobile application. All requests to this server should be authenticated.

Farmers will be able to register their farms within the system, allowing for sheep observers to attribute observations to a specific farm. Furthermore, farmers should be able to generate a sheep supervision report at the end of the grazing season.

1.2 Stakeholders

In order to create a successful system, one must first consider the different stakeholders who are intended to interact with it. This subsection details the different stakeholders surrounding the project and how the system relates to them.

1.2.1 Farmers

The farmers are the ones who own the sheep, and are also the ones who have the most to lose if a sheep is killed. Farmers are incentivized to keep their sheep alive through several factors, whether they are economical, legal or emotional. The main priority of most farmers is to ensure the well-being of their sheep; They are therefore interested in tooling providing them a greater overview of the threat level in their grazing areas. The proposed solution could allow for farmers to gain a greater overview, both through a greater amount of data to analyze, and by optimizing the sheep observation trips so that they can be performed more often.

1.2.2 Sheep Observers

The sheep observer is the end user of the mobile application. One of the intended benefits of digitalizing the documentation process is the simplification and optimization of sheep observation registration. The sheep observer is mostly concerned with ease of use and reliability. Old methods are tried and true; In order to allow for a smooth transition to a new way of registering observations, the benefits must be immediately noticeable.

1.2.3 County Governor

The county governor wears many hats. The one most relevant for this project, however, is the reimbursement of sheep that have been killed by predators. In order to do this, the county governor needs a solid foundation of data that can confirm claims made by a farmer.

1.2.4 Mattilsynet

The Norwegian Food Safety Authority, Mattilsynet, is a Norwegian organ concerned with the quality of Norwegian-made food and the well-being of Norwegian farm animals. As such, Mattilsynet is interested in accessing data regarding sheep supervision. Having access to consistent and detailed data would simplify the job Mattilsynet has to do.

1.2.5 Statens Naturoppsyn

The Norwegian Environment Agency, Statens Naturoppsyn (SNO), is concerned with any sheep deaths that can possibly be attributed to predators. Whenever grazing personnel encounters a sheep that met a suspicious demise, SNO should be called to determine the cause of death. As such, SNO is concerned with receiving reports of dead sheep, their whereabouts, and any other contextual information SNO deems to be relevant.

1.3 Thesis Structure

Due to the nature of this project, the resulting thesis document is structured in an unorthodox manner. Each system component is described in its own part. The components of the resulting system differ in terms of functionality and implementation; Grouping them together would not be sensible. As such, the thesis is structured in the following way:

- **Part 1: Introduction** - Introduces the project, domain knowledge, methodologies, processes, system requirements and an overarching description of the final solution.
- **Part 2: Mobile Application** - Provides a detailed description of the supervision application.
- **Part 3: Server** - Provides a detailed description of the server concerned with persisting supervision data.
- **Part 4: Cloud Infrastructure** - Describes how the system cloud infrastructure is provisioned and deployed.
- **Part 5: Closing Statements** - Concludes the thesis with a shared discussion of the entire thesis, suggestions for future work and a proper conclusion to the project.

Chapter 2

Background

This chapter will provide an overview of how shepherding is performed in Norway, the grazing season and the sheep supervision process. Finally, Section 2.2 will provide an overview of existing solutions that fulfills some of the requirements this project attempts to fulfill.

2.1 Shepherding in Norway

Of all land mass in Norway, only 3.5 percent are considered to be arable land [4]. Despite this, Norway manages to contain a large population of farm animals throughout the entire country. The process of tending to farm animals is greatly alleviated by allowing animals to graze freely on outlying fields when conditions allow for it. Approximately 45 percent of Norwegian soil is considered to be outlying fields, and farmers are encouraged to make use of it [5]. Farmers are estimated to save approximately one billion NOK on fodder every year they utilize outlying fields [6].

2.1.1 Seasonal shepherding

Grazing periods in outlying fields vary across the country. In climates with acceptable weather, such as areas close to the ocean, sheep can be kept outside and active year-round. In harsher conditions, sheep usually graze outlying fields during the warm summer months. This is usually in the range of May to October, although it varies from location to location [7].

2.1.2 Ties

In order for farmers to keep track of how many lambs a specific ewe has, Norwegian farmers mark ewes with what they refer to as ties, a plastic ribbon attached around the neck of the sheep. They are shown in Figure 2.1 The usage of ties allows

for easier supervision during the grazing season, both for the observer and any other parties traversing the area. If the number of lambs grazing with an ewe does not match the amount indicated by the tie, one can assume a lamb is missing. Norsk Sau og Geit (NSG), an organization for Norwegian sheep- and goat-keepers, presented the following standard in 2011 [8]:

- 0 lambs: Red tie
- 1 lamb: Blue tie
- 2 lambs: Yellow or no tie
- 3 lambs: Green tie

The standard is followed throughout the entirety of Norway. Oppland county used to deviate from this standard, but decided to adopt it after the 2019 season [9].

Figure 2.1: Sheep tie colors, as proposed by NSG.

2.1.3 Loss of Sheep

Farmers expect a certain number of sheep to die every grazing season. In the 2019 grazing season, approximately 1 934 000 sheep were released, of which approximately 102 000 never returned [1]. Causes of death varies, but are most commonly attributed to illness, herd separation, accidents and predators.

Of the aforementioned causes, farmers are the most interested in deaths caused by predators. The Norwegian environment is home to several predators, namely the Eurasian Lynx, Brown Bears, Wolves, Wolverines and Eagles. These species are protected under Norwegian law and the Bern convention, and can therefore not simply be killed [10]. However, the Norwegian government wants animals and

predators to live side by side without any major losses.

Thus, a certain number of grazing animals are expected to be killed by predators every year. As a remedy, the government allows farmers to submit reports describing animal deaths that can be directly tied to predators. Farmers are then paid reparations for their losses. However, determining the cause of death can oftentimes be a challenging procedure. For a farmer to be able to even attempt to claim predators as the cause of death, the animal in question has to be located. Predators are known to move their prey, and even hide it at times. Even if one were to discover the animal, attributing the death to a predator can still be challenging. The possibility of the animal dying due to starvation, accidents or illnesses are always present. Due to the challenges surrounding attributing cause of death to predators, the Norwegian government requires all carcasses that could fall within the aforementioned cause of death to be reported to the Norwegian Nature Surveillance (Statens Naturoppsyn (SNO)), whom in turn will decide the cause of death.

2.1.4 Supervision

Norwegian law requires one to perform supervision of grazing animals at least once a week [3]. There are currently no recommendations as to what sheep observers are to do when animals are separated by large distances. As such, farmers and observers are left to handle such problems on their own. Areas with proven predator activity must be visited more often. During these supervision trips, the person conducting the trip must note every sheep observation. This has traditionally been done using pen and paper, although some observers have experimented with digital solutions. For every sheep observation, the observer should document the following attributes:

- The position of the observation (coordinates).
- The time of the observation.
- The number of sheep.
- The number of sheep separated by wool color (black, gray, brown).
- The number of sheep separated by tie color.
- What farms the sheep belong to.

Some of this information can be difficult to observe from a distance. According to Svein-Olaf Hvasshovd, the supervisor of this project and an experienced sheep observer, tie colors and farm colors can normally not be identified if the observer is more than 30 to 50 meters apart from the sheep in question. For distances larger than this, observers are not expected to register tie colors and farms.

Farms usually mark their sheep in such a way that they can easily be identified as theirs. This is usually achieved through the use of ear markings, with two colors

uniquely identifying a farm within the grazing area.

Storing this information allows for the farms to have a certain amount of knowledge with regards to where their sheep have been observed and how many sheep they have lost thus far in the season. The observer can determine the intended size of the flock by using the tie colors, which can then be compared to the actual number of sheep in the flock. Observing an ewe with a green tie and only two lambs could be a cause for concern, as green ties indicates the presence of three lambs.

Dead sheep should also be registered. If one were to encounter such an issue, the color of the sheep tie and details surrounding the ear markings of the sheep should be documented. The carcass should be photographed in detail to allow for one to retain as much information as possible. Additionally, the farmer should notify SNO if the death could have been caused by predators.

Beyond this, there are several other things that are not mandatory, albeit beneficial, to note down. Any observations of predators in the surrounding areas are extremely relevant for every farmer that has their animals grazing in proximity to the predator sighting. If one were to discover tracks, carcasses, excrement or an actual predator, noting it down could allow for farmers to take proactive actions in order to ensure sheep safety. One could also note down sheep separated from their herd, or destroyed equipment such as bridges and fences.

Finally, when the grazing season ends and the animals are brought back to their farms, the supervision trip notes will be combined into a report for the entire grazing season. A supervision report can either be submitted for individual farmers, or for a combined team of farmers and sheep observers. The resulting report will serve as proof of proper supervision throughout the grazing season. If it is not filled in to the extent the government and Mattilsynet expects, problems regarding reimbursement for lost sheep may occur. Furthermore, lacking documentation can incur strong reactions from Mattilsynet with regards to animal health. Thus, it is very important to keep detailed notes throughout the entire grazing season.

The seasonal reports do not have a standardized format. Instead, the government provides a list of requirements surrounding the information to include. Every trip must include the date it was performed at, the total number of sheep and lamb observed throughout the trip and any occurrences of dead or injured sheep. Furthermore, the government wants every observer to provide as much details as possible with regards to the route they took during the trip. Finally, all observations of potential predators should be included. If an observer spots a predator, it should note down the type of predator and the location of where it was observed.

2.2 Existing Supervision Solutions

Sheep supervision is a narrow area, with little in the way of existing solutions. At the time of writing, no solution fulfills every requirement. Nonetheless, this section will introduce a set of applications that covers some of the functionality required for documenting sheep supervision.

2.2.1 BeiteSnap

BeiteSnap is developed by farmers, and is a cross-platform application for both iOS and Android [11]. The application not only supports digital supervision, but also a plethora of supplementary functionality to allow sheep farmers to be up to date on the state of grazing sheep. Offline maps can be downloaded before starting a trip to ensure that observations can be documented even when internet connectivity is sparse. Furthermore, BeiteSnap can alert other farmers if one of their sheep were found dead. It also provides functionality for generating seasonal reports to be sent to the county governor once the season ends. Screenshots of the application are shown in Figure 2.2.

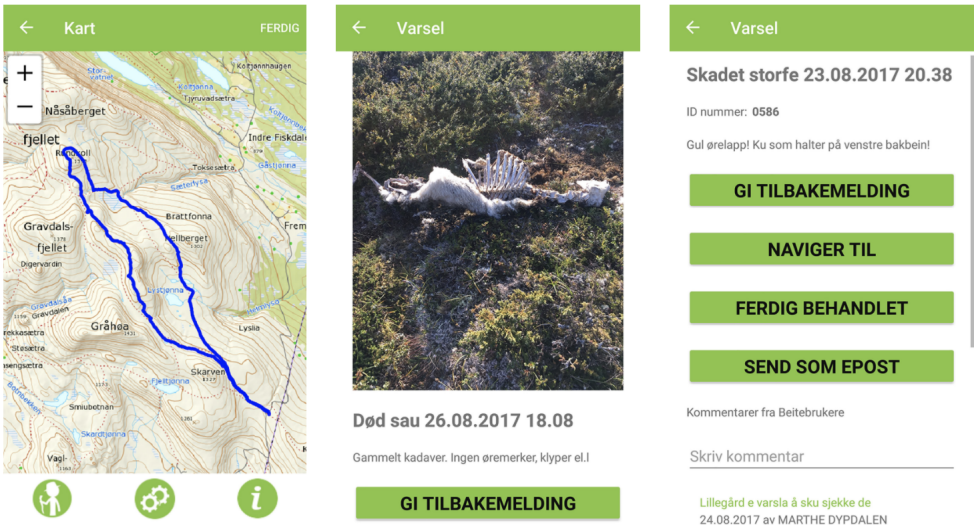

Figure 2.2: BeiteSnap user inteface [12].

Whereas this project mainly focuses on sheep observers hired by farmers, BeiteSnap also focused on private individuals not attached to any specific farm. Through the application, private individuals can alert farmers of dead or lost sheep, allowing for farmers to either fetch their sheep or gain a greater reimbursement from the government.

In terms of functionality, BeiteSnap appears to cover most, if not all, of the required functionality for sheep supervision trips. However, Beitesnap is no longer in

operation. As such, the author is not able to experiment with the application. The information presented in this section is based on the limited information BeiteSnap provided on their website.

2.2.2 Skandobs

Skandobs is an application for registering predator observations in Norway, Sweden, Denmark and Finland [13]. It is available for web, Android and iOS. The application provides functionality for viewing filtered reports of predator observations in any of the supported countries, and whether the observations have been investigated or confirmed. Beyond this, the application allows users to register their observations, in which they can specify where and what they have observed. The registration process also allows for uploading images related to the sighting. Screenshots of the Skandobs user interface are shown in Figure 2.3.

A solution like this is extremely helpful to farmers with regards to being aware of potential threats in their area, but it cannot be used to simplify the supervision process. It is entirely reliant on web connectivity, and it has no support for storing trip data or sheep observations.

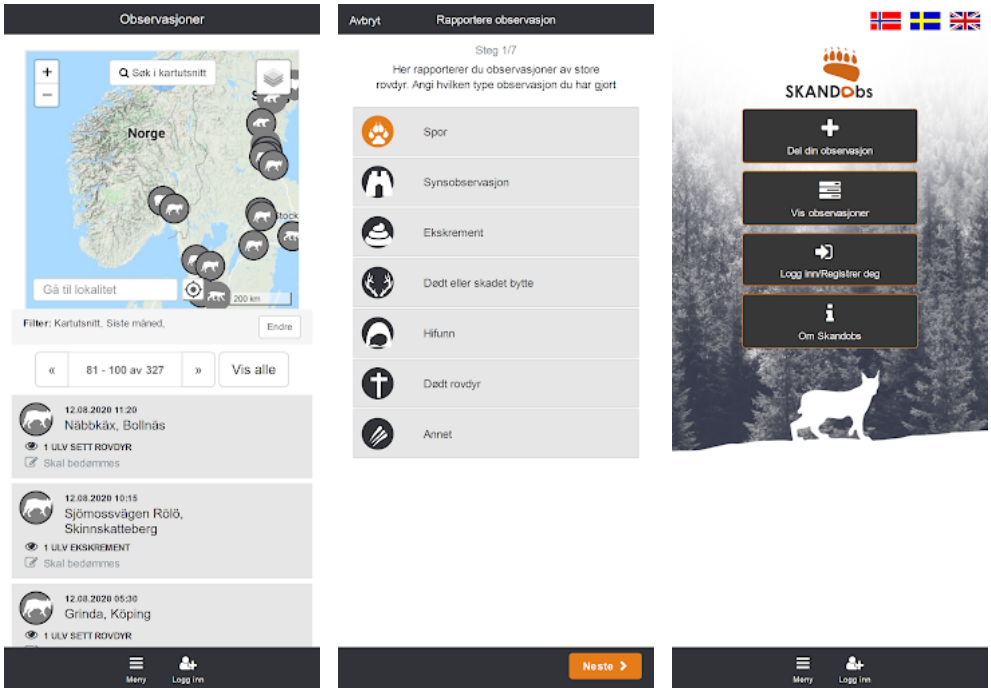

Figure 2.3: Skandobs mobile interface.

2.2.3 Electronic Location Tracking

A popular solution for simplifying the process of sheep supervision is the use of electronic tracking equipment. By fitting sheep with electronic tracking devices, farmers are able to continuously observe their locations. This space is occupied by several actors, with notable examples being Findmy [14], Telespor [15], Nofence [16] and Smartbjella [17]. These solutions provide farmers with electronic tracking devices and applications enabling them to continuously check the whereabouts of their sheep. The aforementioned companies implements their tracking devices as either bells or collars, allowing farmers to place them around the neck of an animal. For reference, an image of the Findmy product is shown in Figure 2.4. Findmy, Nofence and Smartbjella provides the user with the option of creating geofences, a virtual border in which sheep are allowed to graze. If the geofence is crossed, the farmer is notified.

Figure 2.4: The Findmy e-bell model 2 [14].

The tracking devices are highly customizable. Farmers can for instance choose how often location updates should be received, or if the devices should notify the farmer if a sheep has not moved throughout a prolonged time period.

A major benefit of the Findmy product is the way it relays information back to farmers. Telespor, Nofence and Smartbjella rely on LTE-M or Narrowband IoT solutions for transporting data to and from the farmer. This connection does not cover the entirety of Norway. Findmy, on the other hand, relies on a satellite connection, allowing for an increased area in which the tracking devices can function as intended. Predators usually traverse land where there are none or low signs of

human activity. Such areas often correspond to the areas lacking cellular connectivity. If an attack occurs in such an area, the electronic bells and collars would be of little use. However, one should note that most areas are covered; Utilizing electronic tracking devices would, in most cases, provide sufficient tracking.

Utilizing electronic tracking devices allows for farmers and sheep observers to more easily track and manage grazing sheep. However, equipping every sheep with a tracking device is not economically sound. Costs related to tracking devices are shown in Table 2.1. Each device must be purchased at a base price. Furthermore, the farmer must pay a subscription fee for every device they utilize every season. NSG estimates the value of a single sheep to be NOK 3585, whereas a single lamb is worth NOK 1850 [18]. Many farmers are unable to utilize tracking devices due to the associated costs. Some municipalities are willing to subsidize the purchase prices of electronic tracking devices, allowing farmers to utilize them for a certain portion of their sheep. Tracking devices are a great supplement for improving and simplifying the supervision process, but they cannot replace manual observations.

Brand	**Device**	**Subscription**	**Note**
Findmy [19]	1849	229	Quantity discounts.
Telespor [20]	899	149	Several subscription types. Quantity discounts.
Nofence [21]	N/A	799	Devices are rented or leased. Quantity discounts. Pay as you go.
Smartbjella [22]	949	238	Several Subscription types.

Table 2.1: Device and subscription prices for a single tracking device. Prices are in NOK and VAT is excluded. Subscription price is yearly.

2.2.4 Map Applications

The mobile ecosystems provides an abundance of mapping applications through their respective app stores. Many of these applications provides some of the functionality required to create an application like the one this thesis intends to create. By quickly traversing the App Store, one can find map applications that support offline maps, map markings and trip location saving. The Norwegian space, for instance, is occupied by the likes of "Hvor?" [23], "Ut.no" [24], "Topokart" [25] (iOS), "Norgeskart" [26] and "Norgeskart Friluftsliv" [27]. Although these applications fulfill many of the required functionalities, they were developed for a completely different purpose. The aforementioned applications focus on traversing the Norwegian terrain for the purpose of hiking or walking. Map markings are generally used to save fishing spots, cabins or other points of interest. This does not exclude the possibility of using the feature to mark sheep locations. However, if one were to use it to do exactly that, there would not be any standardization

of the noted values, which in turn could lead to a cumbersome process when creating a report. Furthermore, these applications have little to no support for data exportation. Report creation would entail a lot of manual entry in order to have the data in the correct place and format.

Chapter 3

Requirements

In order for a solution to be deemed valuable, a set of high-level requirements would have to be fulfilled. These requirements were already well-defined for this project, due to the supervisor's extensive knowledge of the problem area. This chapter describes the process of eliciting the requirements for the project, followed by descriptions of the use cases the author created based on the available information. The final subsections will then describe the functional and non-functional requirements of the system in detail.

3.1 Elicitation

Throughout previous research and experience within the field, the project supervisor has gained an extensive amount of information surrounding the problem area. This information was provided in the description of the project itself, in discussions between the author and the supervisor before the project started, and in the initial supervisor meetings. However, such broad requirements oftentimes misses minute details, and should never be considered to be a final list of requirements.

The meetings, and the provided high-level requirements, provided a great baseline for the collection and refinement of the project requirements. Initial requirements were translated into high-level use cases, which in turn would be broken down into smaller, more defined, functional requirements. This process also resulted in a set of quality attributes aligning with the goals of the project.

Although the aforementioned method produced many requirements, additional requirements would be added throughout the entire SDLC by using several elicitation methods. Prototyping proved to be the most efficient method. Supervision meetings were held weekly, and allowed the author to regularly present new prototypes. The prototypes usually consisted of abstract solutions or actual code implementations, and allowed for the supervisor to give feedback on any additional functionality or

requirement the prototype needed. This agile and iterative process allowed for the functional requirements to be refined throughout the entire development period, and gave the author a clear sense of what functionality to prioritize.

Usability testing proved to be another integral part of eliciting requirements. Throughout the entire SDLC, the author continuously performed informal usability tests on potential prototypes. This often led to changes in the user interface of the application, and even introduced new requirements or functionality at times.

3.2 Architecturally Significant Requirements

System requirements can be separated into several different categories. This section provides a list of system requirements significantly impacting the overall architecture of the system. Each sub-section will present a requirement and detail how the architecture was adapted to address the requirement.

3.2.1 Offline Capability

Being able to utilize the application without internet connectivity is the most important requirement to consider when planning the system architecture. Whenever an observer conducts a supervision trip, there is no guarantee of internet connectivity. As such, the mobile application would have to somehow store all required data locally on the device. Furthermore, the server or database must be able to support synchronization in some capacity.

3.2.2 Cross-platform application

Many popular mobile applications are available for both iOS and Android devices, a requirement also faced by this project. This could either be achieved through a cross-platform framework or by creating two native implementations. Web applications are not a viable option, as sheep supervision requires system functionality not available through mobile web browsers. The different approaches will drastically alter the architecture and development process of the mobile application, although the system as a whole largely remains unchanged.

3.2.3 Authentication & Authorization

In order to limit resource access to the users who should be permitted to access them, the system must allow for actors to identity themselves through authentication. By introducing authentication, the system will also have to support account registration and account deletion.

3.2.4 Server capabilities

The data generated through the application becomes more valuable when it can be shared with others. Allowing the user to extract this data from the device through tools would not be very practical, as the sharing process would be too complicated. As such, a centralized storage solution would be beneficial. Thus, the planned architecture must consist of some type of central server and storage solution.

3.3 Use Cases

The use cases presented in this section describes the core functionality of the application, and were mostly generated from data presented in the initial weeks of the project. If a new requirement appeared throughout the development period and was deemed to be too large, a new use case would be created and broken down into smaller requirements later-on. The following subsections will provide pictures and descriptions of every use case.

3.3.1 Authentication

Figure 3.1 displays a set of use cases relating to user authentication. The use case features two actors, namely the "user" and the "identity provider". A user within this context can be any actor utilizing the system to some extent, whereas the identity provider is an actor concerned with authenticating and authorizing users.

The "log in" use case indicates that the system should support authentication. If a user has forgotten their password, the "reset password" use case specifies that one should be able to create a new one. Account creation should be performed in two steps; Specify login credentials, which will be stored with the identity provider. Then, specify general user details for usage within the system.

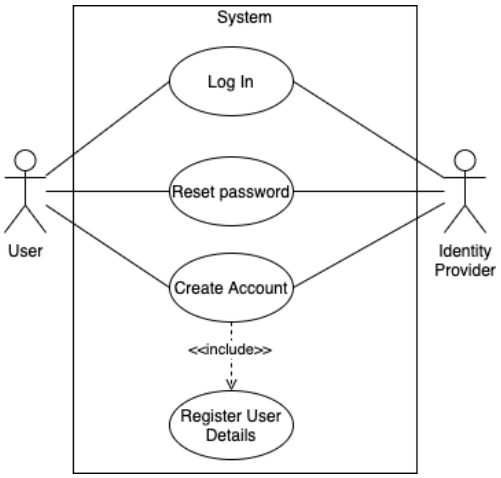

Figure 3.1: Use cases related to authentication.

3.3.2 Download Map

Downloading map portions are an integral part of the application, as it allows sheep observers to utilize the application without any need for internet connectivity. The "download map" use case, as shown in Figure 3.2, specifies the steps one must take to download a map for offline usage. The "sheep observer" actor first has to view the map, and navigate to the specific portion of interest. Then, the actor must select the portion of the map to be downloaded. Before the download can begin, the actor must specify a name for the map to be downloaded. Finally, map tiles are downloaded from the "tile server" actor and stored on the device.

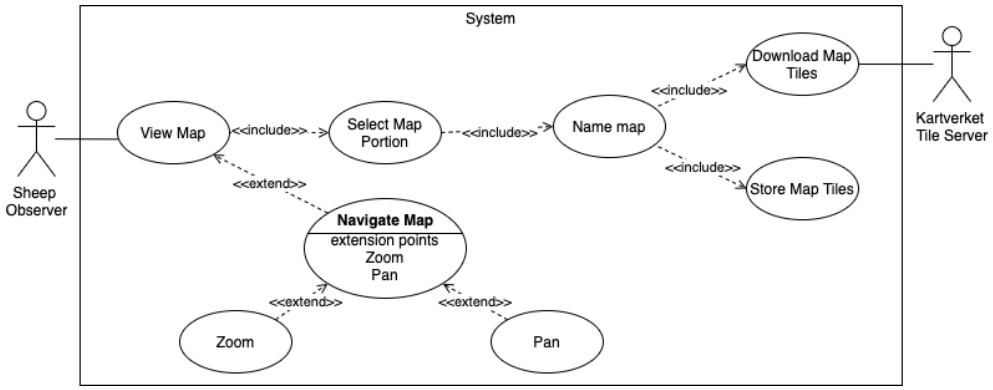

Figure 3.2: Use case diagram for downloading a map.

3.3.3 View Downloaded Maps

Figure 3.3 specifies all the ways one can interact with a downloaded map. One should be able to delete maps, as specified by the "delete map" use case. Furthermore, users should be able to view downloaded maps, as specified by the "preview map" use case. Finally, downloaded maps should allow a user to start a new trip within a specific map. If one is to start a new trip, the observer must first specify what team the trip should be registered under.

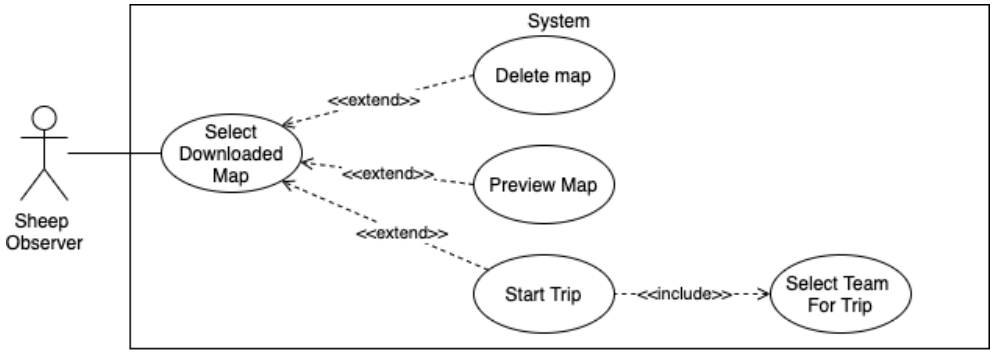

Figure 3.3: Use case diagram for viewing downloaded maps.

3.3.4 Team Actions

Teams actions can be described as everything one is allowed to do when interacting with a specific team, and are shown in Figure 3.4. The user should be able to perform different actions depending on what role they have within a specific team. A team owner should be able to view pending invites, add and remove both members and invites, and delete the entire team. Furthermore, owners should be able to transfer ownership rights to another team member. Team members, on the other hand, should only have access to a subset of the available actions, namely viewing what trips the team have conducted, and leaving the team. These actions should also be available to the team owner, although leaving a team has the prerequisite of having to transfer ownership rights before doing so.

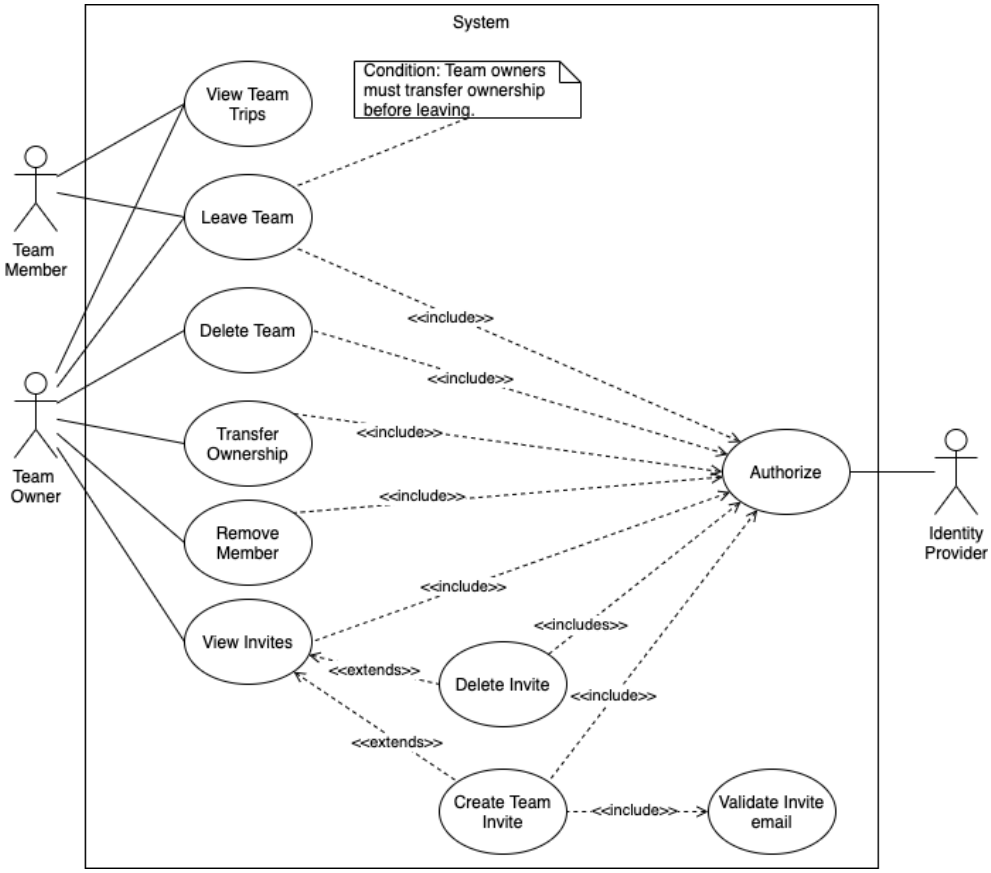

Figure 3.4: Use case diagram showing available actions for a team.

3.3.5 New Team

Figure 3.5 shows the use case of creating a new team, in which a user specifies a team name and an optional description.

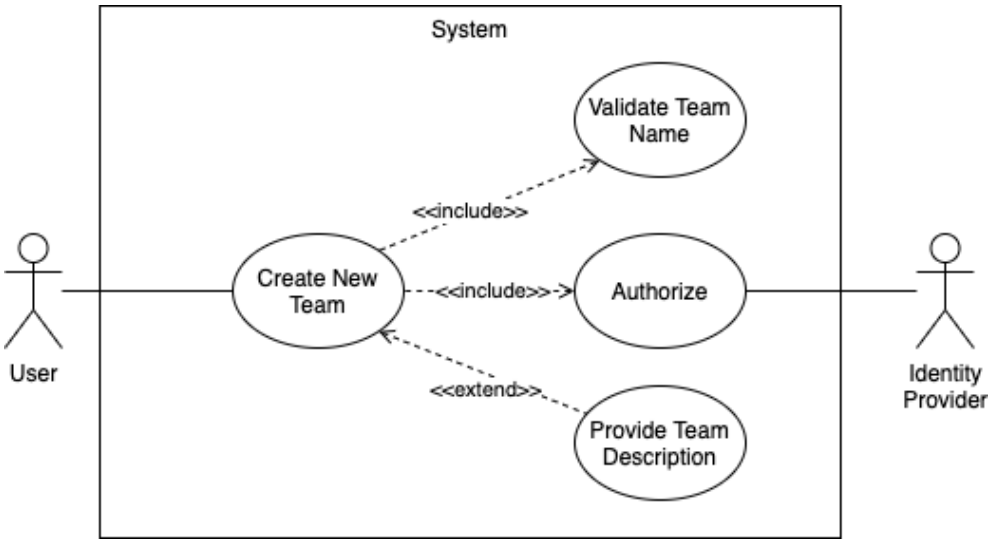

Figure 3.5: Use case diagram for creating a new team.

3.3.6 View User Invites

The use case of viewing invites are displayed in Figure 3.6. Users should be able to either accept or decline an invite.

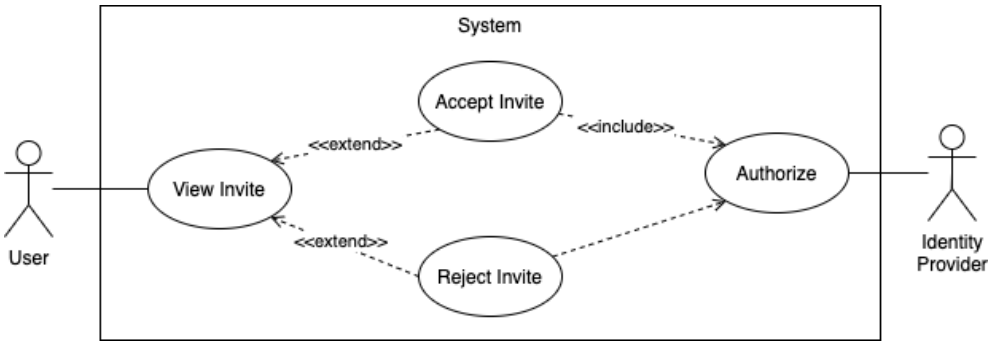

Figure 3.6: Use case diagram for managing user invites.

3.3.7 Farm

The system should allow farmers to register their farms for offline usage during supervision trips. Figure 3.7 also lists the possibility to delete farms, which are accessed by both sheep observers and farmers. If a farmer chooses to delete their farm, it will be deleted from the entire system. Sheep observers, on the other hand, only delete locally stored instances of a farm. In the rest of the system, the farm remains intact.

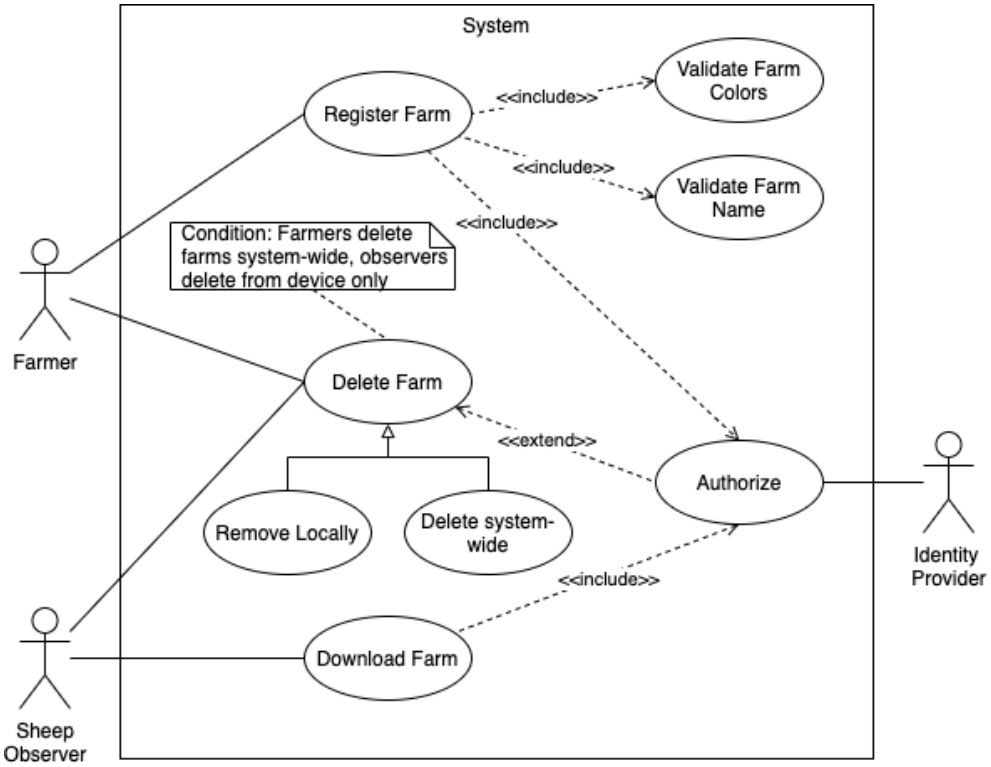

Figure 3.7: Use case diagram for farm interactions.

3.3.8 Perform Trip

The use case diagram in Figure 3.8 details the available actions when conducting supervision trips. One can either choose to perform a trip by starting a new trip, or by resuming a previously started trip.

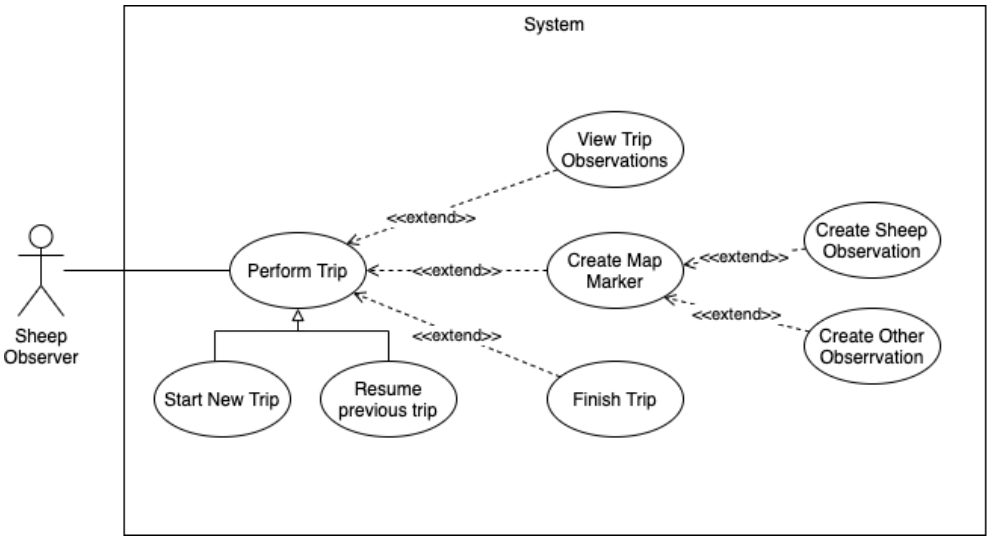

Figure 3.8: Use case diagram for performing a trip.

Once the trip has started, the user can continuously create map markers, which allows one to mark where an observation has taken place. This observation can either be a sheep observation, or an "other" observation. Furthermore, the user should be able to view previously registered observations for the ongoing trip. This use case is further expanded upon in Section 3.3.11. Finally, the user should be able to finish the trip at any time.

3.3.9 Perform Sheep Observation

Figure 3.9 breaks down the use case of performing sheep observations. The observer is required to register the total number of sheep for every observation, as well as the number of sheep divided by the color of their wool. If the observation is within 30 to 50 meters of the observer, one should be able to specify the number of each tie color as well. Finally, close observations should allow the observer to register the farms sheep belongs to. Registering farms should allow the observer to both specify downloaded (known) farms and unknown farms.

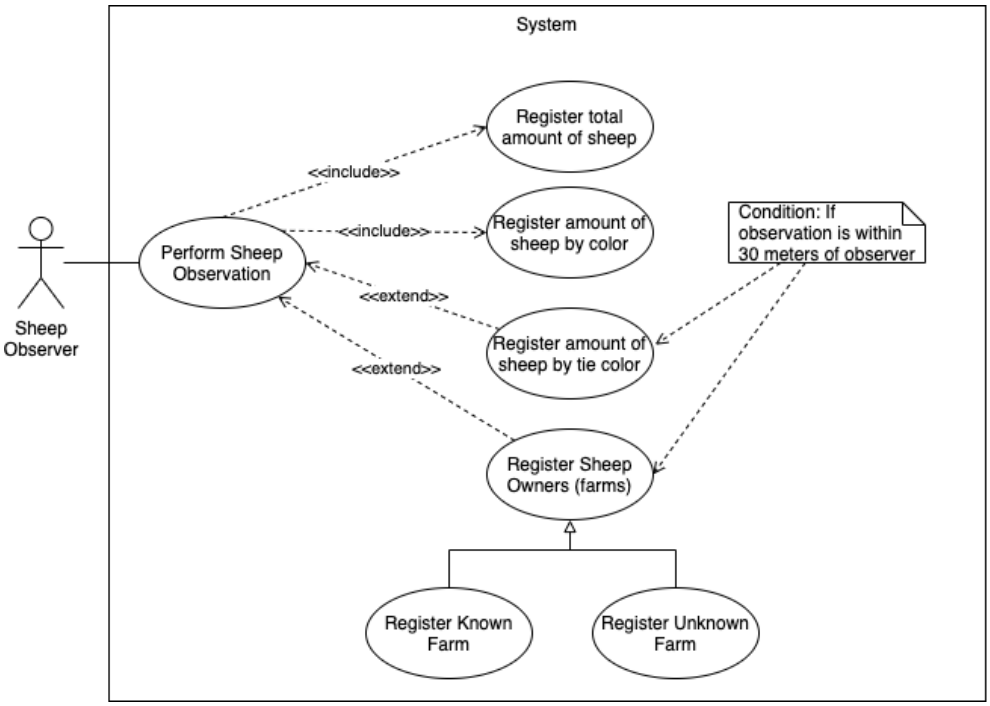

Figure 3.9: Use case diagram for performing a sheep observation.

3.3.10 Perform Other Observation

Any type of observation not related to alive sheep falls under the "other" category, and should follow the same registration procedure. Every observation must specify the type of observation, and a succinct description of what has been observed. One can also optionally add observation images, whom in turn can have optional image descriptions. The use case is shown in Figure 3.10.

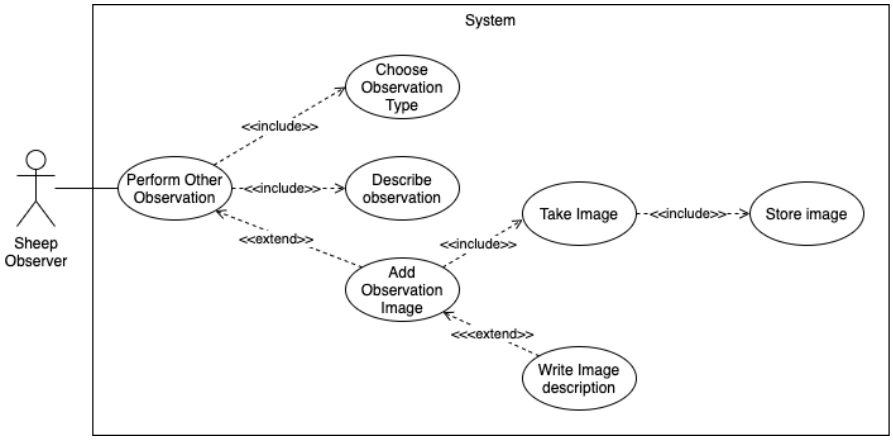

Figure 3.10: Use case diagram for performing an "other" observation.

3.3.11 View Previous Trips

The use case of viewing previously performed trips is shown in Figure 3.11. When viewing previous trips, the user can choose to view details for a specific trip, or delete it entirely. Trips that are not synchronized to the server yet can simply be deleted from the device. Synchronized trips, on the other hand, must be deleted from the server as well. By pressing a specific trip, the user should be able to view details about the trip observations.

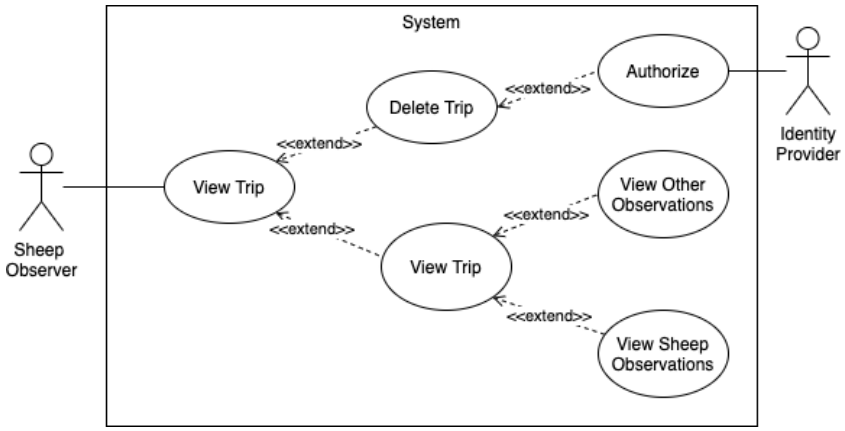

Figure 3.11: Use case diagram for viewing previously performed trips.

3.4 Functional requirements

This section will describe the functional requirements of the system: Requirements describing what type of functionality the system must provide. As stated in Section 3.1, the functional requirements were elicited through several means, namely prototypes, usability tests, the supervisor and breaking down use cases. Functional requirements are listed in Tables 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 3.10, 3.11 and 3.12.

ID	Priority	Description
FR1.0	High	The system should allow users to log in with an email and a password.
FR1.1	High	The system should allow users to create accounts with an email and a password.
FR1.2	High	The system should persist a user log-in as long as the session is valid.
FR1.3	High	The system should allow users to reset their password through email recovery.
FR1.4	High	The system should allow a user to log out.
FR1.5	High	The system should allow a user to delete their account and any personally identifiable information.

Table 3.1: Authentication requirements.

ID	Priority	Description
FR2.0	High	The mobile application should allow for users to register their first and last name.
FR2.1	Medium	The mobile application should not grant users access to the application before first and last name is registered.
FR2.2	Low	The mobile application should allow for users to change their first and last name.
FR2.3	High	The mobile application should allow for users to view their user details.
FR2.4	Low	The mobile application should allow for users to view the first and last name of other team members.

Table 3.2: User details requirements.

ID	Priority	Description
FR3.0	High	The system should allow farmers to register their farms.
FR3.1	High	The system should allow for farmers to delete a registered farm.
FR3.2	High	The system should allow for farmers to update their registered farms.
FR3.3	High	The mobile application should allow users to search for registered farms.
FR3.4	High	The mobile application should allow for users to download farms for offline usage.
FR3.5	High	The mobile application should allow for users to view downloaded farms.
FR3.6	High	The mobile application should allow for users to remove downloaded farms.
FR3.7	High	The mobile application should allow for users to specify unknown farms that have not been downloaded.

Table 3.3: Farm requirements.

ID	Priority	Description
FR4.0	High	The system should allow users to send and receive team invitations.
FR4.1	High	The mobile application should allow for users to view their pending invitations.
FR4.2	High	The mobile application should allow for users to accept or decline a pending invite.
FR4.3	High	The mobile application should allow team owners to send team invites to other users.
FR4.4	Medium	The mobile application should allow team owners to delete pending invites for a team.
FR4.5	Medium	The mobile application should allow for team owners to view a list of pending invites for a team.

Table 3.4: Invitation requirements.

ID	Priority	Description
FR5.0	High	The system should allow users to create a new team.
FR5.1	High	The mobile application should allow users to define a name and description for new teams.
FR5.2	High	The mobile application should allow for users to view the teams they are members of.
FR5.3	High	The mobile application should allow for users to view the details of a specific team.
FR5.4	High	The mobile application should allow for users to view the members of a team they are a member of themselves.
FR5.5	High	The mobile application should allow for users to leave a team.
FR5.6	Medium	The mobile application should allow team owners to delete their team.
FR5.7	Low	The mobile application should allow team owners to change the name and description of a team.
FR5.8	Medium	The mobile application should allow team owners to remove team members.
FR5.9	Medium	The mobile application should allow team owners to transfer ownership privileges to another member.
FR5.10	High	The mobile application should allow users to view their teams without internet connectivity.

Table 3.5: Team requirements.

ID	Priority	Description
FR6.0	High	The mobile application should allow users to view the trips they have performed.
FR6.1	Medium	The mobile application should allow users to view trips performed within a specific team.
FR6.2	High	The mobile application should allow users to delete trips they have conducted themselves.
FR6.3	Medium	The mobile application should allow users to view team trips without internet connectivity.
FR6.4	High	The mobile application should allow users to view their own trips without internet connectivity.
FR6.5	High	The mobile application should allow users to view all observations for a specific trip.
FR6.6	High	The mobile application should allow users to send their own trips to the server.
FR6.7	High	The mobile application should only allow users to delete observations for trips not yet sent to the server.

Table 3.6: Trip management requirements.

ID	Priority	Description
FR7.0	High	The mobile application should be able to display a map to the user.
FR7.1	High	The mobile application should allow for users to pan the map by dragging.
FR7.2	High	The mobile application should allow for users to zoom the map by pinching.
FR7.3	High	The mobile application should allow for users to download specific areas of the map for offline use.
FR7.4	High	The mobile application should allow for users to specify a name for a downloaded map area.
FR7.5	High	The mobile application should allow for users to view a list of downloaded map areas.
FR7.6	Medium	The mobile application should allow for users to delete a specific map area.
FR7.7	High	The mobile application should allow for users to view a downloaded map area.
FR7.8	High	The mobile application should display a marker specifying the current location of the user on the map.
FR7.9	Low	The mobile application should allow for users to rename a downloaded map.

Table 3.7: Map requirements.

ID	Priority	Description
FR8.0	High	The mobile application should allow for users to start a trip for a specific team.
FR8.1	High	The mobile application should continuously track and store the user position throughout a trip.
FR8.2	High	The mobile application should allow for users to view observations for an ongoing trip.
FR8.3	High	The mobile application should allow for users to delete observations during an ongoing trip.
FR8.4	High	The mobile application should allow for users to register observations on a specific map point.
FR8.5	High	The mobile application should allow for users to finish a trip.
FR8.6	Medium	The mobile application should allow for users to abort an ongoing trip.
FR8.7	High	The mobile application should allow for users to resume an ongoing trip.
FR8.8	Medium	The mobile application should allow for users to view the distance between themselves and a potential observation.
FR8.9	High	The mobile application should allow for users to view their route for an ongoing trip.
FR8.10	High	The mobile application should allow for users to conduct trips without internet connectivity.

Table 3.8: Trip requirements.

ID	Priority	Description
FR9.0	High	The mobile application should allow for users to register the total number of sheep in a sheep observation.
FR9.1	High	The mobile application should allow for users to register the number of sheep with a specific wool color in a sheep observation.
FR9.2	High	The mobile application should allow for users to register the different tie colors in a sheep observation if the observation is less than 30 to 50 meters away from the observer.
FR9.3	High	The mobile application should allow for users to add tie colors without having to look at the screen.
FR9.4	High	The mobile application should allow for users to register what farms are present in a sheep observation if the observation is closer than 30 to 50 meters away.
FR9.5	High	The mobile application should allow for users to add unknown farms to the list of observed farms for a sheep observation.

Table 3.9: Sheep observation requirements.

ID	Priority	Description
FR10.0	High	The mobile application should allow users to specify the type of observation being made.
FR10.1	High	The mobile application should allow for users to describe the observation.
FR10.2	High	The mobile application should allow for users to add an optional amount of pictures to the observation.
FR10.3	Low	The mobile application should allow for users to add a description for each picture in the observation.

Table 3.10: Other observation requirements.

ID	Priority	Description
FR11.0	High	The mobile application should be able to automatically synchronize locally stored content to reflect the state of the server database.
FR11.1	Medium	The mobile application should allow users to manually request server synchronization.

Table 3.11: Synchronization requirements.

ID	Priority	Description
FR11.0	High	The system should allow for users to generate supervision reports based on supervision data at the end of a season.
FR11.1	High	Generated supervision reports should be available for download as a PDF.

Table 3.12: Report generation Requirements.

3.5 Quality attributes

Whereas Section 3.4 describes what the system must be capable of doing in terms of functionality, this section describes what qualities the entire system should strive to uphold. The following sections will describe the quality attributes chosen for this project, namely Availability, Modifiability, Usability and Security.

3.5.1 Availability

The Norwegian environment vastly differs throughout the country, and so does the cellular reception. In order to ensure high availability, one must plan and design with this caveat in mind; Internet connectivity is not guaranteed, and should not be expected. This property aligns well with the description of availability, a quality attribute focused on ensuring that software is usable whenever one needs it to be [28].

The best way to avoid catastrophic failures is to expect them. In the case of this application, relying on internet connectivity would almost certainly result in failure at some point during the grazing season. By anticipating the lack of internet, the application would be more tolerant to failures. Tables 3.13 and 3.14 describes two scenarios for the availability attribute.

ID	**A1**
Source	Application
Stimulus	Unable to achieve an internet connection
Artifact	Application
Environment	Degraded operation
Response	User is able to start, execute and finish a trip
Response measure	Trip is completed without any technical problems

Table 3.13: Availability scenario 1: Executing a trip without internet connectivity.

ID	**A2**
Source	Load Balancer
Stimulus	Unable to reach health check endpoint
Artifact	Server
Environment	Normal operations
Response	Load Balancer redirects traffic to another compute instance
Response measure	Without any downtime

Table 3.14: Availability scenario 2: Verifying load balancing capabilities.

In order to ensure high availability, the following availability tactics will be utilized:

Monitor

Ensuring the well-being of an entire system is oftentimes complicated. Bass et al. suggest employing an entire component solely for this purpose, namely the monitor. They go on to describe a monitor as a "... component that is used to monitor the state of health of various other parts of the system..." [28]. Many cloud providers offers monitors as a service, providing easily configurable monitoring for crucial resources. In the case of this project, monitors will be used to both route traffic to and inspect the health of the computational instances the server runs on.

Ping / Echo

The aforementioned monitor is capable of checking the health of a specific component. In order to achieve this, it requires a way of ensuring that the monitored component responds as intended. One way of doing this is by providing the monitor with the knowledge of how to reach a specific portion of the monitored component, and describing what the portion should return under normal circumstances. This process is the "Ping / Echo" tactic; If the monitored component does not respond as expected, the monitor will know that the component is unhealthy [28]. The tactic will be used to monitor the health of the system servers.

Exception Detection

Bass et al. defines exception detection as "the detection of a system condition that alters the normal flow of execution" [28]. Modern programming languages provides a plethora of tools for detecting such errors even before code is ran by continuously scanning the code for errors. By detecting possible exceptions the system can encounter with the written code, the developer is able to either preemptively handle them or change the code altogether. Another strategy many programming languages provide as a built-in feature is static typing. When calling

a function, the type defined in the function signature must align with the variables one attempts to pass to it. Using static typing allows for a more rigid final product.

Another strategy within this genre is the use of timeouts. Whenever it becomes apparent that a service does not return a response within an allotted time slot, it should be canceled. The application developed in this project utilizes both sensors on the smart phone and external servers, whom are all capable of failing at any time. Being able to cancel a failed request if it takes too long, for instance, would allow the application to go back to a functioning state instead of indefinitely waiting for the failed request to finish.

Exception Handling

In stark contrast to popular belief, failure is always an option. Applications can fail at any time, and can be caused by a near unlimited amount of reasons. In order to provide a rigid application that works as intended, one must account for such failures and handle them appropriately [28]. The planned application is dependent on several smartphone-sensors and external resources in order to function as intended. One cannot simply expect these to work as intended at all times. Catching and handling such exceptions allows the application to revert to a functioning state, and is critical in order for the application to be useful.

Rollback

As previously mentioned, failure is imminent. One way to handle failures is to revert any actions that occurred before the failure, allowing the application state to remain "correct" [28]. To apply the strategy to the realm of this project, one could for instance imagine functionality for storing something to two different database tables. If the second insertion fails, and the first relied on the second, the database would be in an invalid state. In such a case, the first table should be reverted to the previously known "correct" state. By allowing the application to always remain in a "correct" state, one mitigates the chance of a critical failure.

Retry

If a specific part of an application or system fails, an approach to fixing the failure is to try again in hopes of a success [28]. Failures can have temporary causes, and the process of simply trying again can be effective at times. In the context of this project, the retry tactic will be used when the application communicates with remote resources.

Degradation

The continuous threat of losing internet connectivity is one of the most important factors to consider during development. As such, some sort of alternative to remote

storage must be easily available. This application solves this by managing critical application features in an "offline-first" approach; The "degraded" state of the application is the normal state of operation. This fits well under the tactic of degradation, in which the most important functionality of the system in question remains operational regardless of any shortcomings of the environment it runs in [28]. By having an alternative to failing components, the chance of critical failures is drastically reduced.

Transactions

Bass et al. states that transactions are used to ensure all communications comply with the ACID properties [28]. The properties were defined by Haerder and Reuter [29], and are described in Table 3.15.

Name	**Description**
Atomicity	A transaction must succeed completely or not succeed at all.
Consistency	A transaction can only commit valid results.
Isolation	A transaction cannot be aware of the details of other transactions.
Durable	Committed results must be able to survive future failures.

Table 3.15: Descriptions of the ACID properties [29].

Transactions are most commonly used within the context of databases, in such a way that it eliminates the possibility of encountering race conditions. This ensures a consistent result, no matter how many times a set of operations run. The rollback tactic, as defined in Section 3.5.1, is dependent on transactions in order to function. In the case of this application, the tactic will be used to ensure that whatever is stored to the database is stored correctly.

3.5.2 Modifiability

As previously stated, this project is heavily reliant on utilizing prototypes to ensure the best possible user experience. In order to allow for easily creating or modifying prototypes, one should strive to create an easily modifiable system. This is encapsulated within the modifiability attribute, which describes the ease of which a system can be altered [28]. Some modifiability scenarios are described in Tables 3.16 and 3.17. The following sections will describe the modifiability tactics the system employs.

ID	**M1**
Source	Developer
Stimulus	Wishes to replace the current HTTP package
Artifact	Application HTTP service
Environment	Design time
Response	HTTP package is replaced, HTTP service is updated and tested
Response measure	In 4 hours without introducing any side effects.

Table 3.16: Modifiability scenario 1: Replacing the HTTP package.

ID	**M3**
Source	Developer
Stimulus	Wishes to add new query parameters to trip search
Artifact	Server code
Environment	Design time
Response	Query parameter is implemented, tested and deployed
Response measure	Within 2 hours without introducing side effects

Table 3.17: Modifiability scenario 2: Adding a new query parameter.

Split Module

Bass et al. suggests that a larger module oftentimes imposes a greater modification cost than several small modules [28]. This sentiment is shared in others bodies of work within the field, with a notable example being Robert C. Martin's "Clean Code" [30]. Whenever a module grows too large, the developer should consider whether it should be split into several smaller modules where each module has high cohesion within.

Increase Semantic Coherence

The tactic of increasing semantic coherence describes how all functionality within a module should be concerned with the same area [28]. If something breaks this rule, the functionality in question should be moved to another module. By having modules operate on a specific area, the likelihood of encountering side effects when altering a module can be drastically reduced.

Encapsulation

Another well-known tactic to ensure modifiability is encapsulation. By only exposing a certain amount of functionality through an API, the risk of changes within a module carrying over to other modules is greatly reduced [28]. Furthermore, encapsulation allows for the developer to constrain what functionality can be

accessed from outside the module, allowing the developer to hide functionality not intended for outside usage.

Intermediaries

Bass et al. suggests utilizing intermediaries to break dependencies between responsibilities [28]. If an action requires one responsibility to be performed before another, an intermediary will be able to entirely remove the knowledge of the other action, thus allowing for lower coupling and higher cohesion. The system will employ intermediaries whenever a set of operations must be orchestrated in a specific manner.

Refactor

Throughout the development process, one usually develops a greater understanding of how the solution should be built, oftentimes resulting in a different solution than originally planned for. The process of refactoring code allows for a developer to change certain portions of a code base to better align with what is considered to be the best solution to a problem. Within any agile project, the size or purpose of a module usually grows or changes. Refactoring allows for the developer to extract functionality from a growing component into a new component, allowing for a higher cohesion within. The prototype-intensive nature of this project would greatly benefit from refactoring once a final prototype has been settled upon.

3.5.3 Security

In the current age of computing, most systems store some type of information, be it sensitive or non-sensitive. No matter what, developers should always strive to keep information out of the hands of unauthorized actors. Bass et al. describes security as "a measure of the system's ability to protect data and information from unauthorized access while still providing access to people and systems that are authorized" [28]. Ignoring security is a crucial mistake, and can make entire systems collapse. Data breaches, deletions or modifications can cause major disruptions within an organization, and occurs frequently. This section will describe what tactics this project will utilize to properly secure application data and its integrity. Tables 3.18 and 3.19 describes two security scenarios.

ID	**S1**
Source	Unauthorized Attacker
Stimulus	Access system database
Artifact	System database
Environment	Online, normal operations
Response	Rejected attack
Response measure	Rejected 99.99 % of the time

Table 3.18: Security scenario 1: Rejecting unauthorized database requests.

ID	**S2**
Source	Rogue farmer
Stimulus	Delete competing farm logged in as him or herself
Artifact	Server
Environment	Online, normal operations
Response	Request is rejected, unauthorized for action
Response measure	Rejected 99.99 % of the time

Table 3.19: Security scenario 2: Rejecting unauthorized actors.

Detect Service Denial

A common attack vector today is to bring crucial system components down by flooding it with requests, a process known as a distributed denial of service (DDoS). Detecting such attacks can be done by continuously checking network traffic patterns with previously known attack patterns, filtering out all attack-related traffic [28]. This would allow the attacked service to remain up, mostly unaffected by the overflow of incoming requests. In the case of this project, such functionality is handled by AWS, the cloud provider hosting the server.

Identify Actors

The tactic of identifying actors is a crucial one in the modern world of computing. Bass et al. specifies that the tactic "... is really about identifying the source of any external input to the system" [28]. When the alternative is to simply trust every actor interacting with the system, one cannot ignore a tactic such as this. By requiring actors to provide some sort of identifier for every system interaction, the system enables functionality such as access logs, data restriction and permission restrictions.

Authenticate Actors

Identifying actors is of no use unless they are authenticated. By requiring authentication through the use of a secret, impersonating the identity of another person

becomes much more difficult. Every sheep observer and farmer wishing to utilize the system will have to be authenticated.

Authorize Actors

All actors are not equal; Just because an actor owns one resource does not mean they can mutate another. By implementing any type of access control, the system is able to allow or reject requests based on the actions and resources the actor in question is allowed to interact with.

Limit Access

A very important factor to consider when designing a system is the attack surface one exposes. The tactic of limiting access specifies that one should strive to only allow access in a secure manner [28]. By minimizing the possible attack surface of an application, the risk of being breached is significantly lowered. In the case of internet-facing resources, developers might apply restrictions as to where and how the resources can be interacted with, minimizing the risk of unauthorized access. The backend of this system, as described in Chapter 21, severely restricts access to server instances and the database.

Change Default Settings

Many Software as a Service (SaaS) and open source solutions are preconfigured with default credentials, of which many are commonly known across the internet [28]. By not changing default credentials, an easily exploitable attack vector is left open. Changing these credentials to hard-to-guess passwords, or other secure means of access, drastically improves the security of a system.

3.5.4 Usability

In order for a system to reach market acceptance, it has to provide a more enjoyable experience than existing solutions. Any system should be intuitive, efficient and easy to use, traits the Usability attribute embodies perfectly [28]. Usability scenarios are described in Tables 3.20 and 3.21. The following subsections will describe the tactics this project utilized to create a user-friendly application.

ID	**U1**
Source	Sheep Observer
Stimulus	Wants to cancel an in-progress map download
Artifact	Mobile Application
Environment	Runtime
Response	Map download is canceled
Response measure	Canceled within one second

Table 3.20: Usability scenario 1: Cancellation of long-running task.

ID	**U2**
Source	Sheep Observer
Stimulus	Learn to use the mobile application
Artifact	User interface
Environment	Runtime
Response	Navigates application without problems
Response measure	Within 30 minutes

Table 3.21: Usability scenario 2: Application familiarization.

Rapid Prototyping

The project thesis has already mentioned the use of prototyping several times. Creating good user interfaces is an iterative process, and requires a substantial amount of experimentation. By continuously gaining user feedback on what is considered to be good and bad, the developer is able to create effective, easy to use and enjoyable user interfaces [28].

Support User Initiative

In order to provide a good user experience, an application must allow for the user to use it effectively, whilst also expecting and allowing user errors. Users should be able to steer the application in whichever direction the user requires. This project plans to do this by implementing the cancel and aggregate tactics [28].

Wherever an action expects some sort of user initiative, the ability to cancel the action should also always be present. The context of the action in question is usually negligible; It could be caused by user error, a change of heart or any other factor. Furthermore, many applications usually allows for the addition and removal of items from a list. The tactic of aggregating actions would allow a user to select several items at once, and then perform the action on every selected item, optimizing the time the action takes.

Support System Initiative

Any experience with an application is a two-way street. The user provides input, and the system processes it and acts accordingly. In order to maximize the smoothness of this interaction loop, the application should have some built-in method of understanding what the user is attempting to do. This is usually achieved by maintaining models of the task the user is attempting to perform, or a model of the overall system [28].

In the first case, the task can be developed to have some contextual knowledge of what the task requires. For instance, then the application requires the user to fill out a form, it would be beneficial for the user if form fields could provide automatic formatting of the input and validate if it is accepted.

On a larger scale, the system could for instance provide the user with feedback on what it is currently doing. Long-running tasks could provide progress indicators, whereas failed tasks could provide a notice regarding why it failed.

Chapter 4

Development Process

Developing a product is no easy task. In order to boost the chances of creating a successful product, one must be able to properly plan and execute the process. This chapter describes the development methodology utilized throughout the project. Furthermore, it will describe the process of managing tasks and the regular day-to-day of the development process.

4.1 Software Development Methodology

The process of managing how software is developed is immensely important. Everyone involved needs to have a shared vision of the work required to complete the project. In order to effectively perform such tasks, a set of software development methodologies emerged. Previous software endeavors have shown that the wrong choice of methodology could lead to a failed Software Development Life-Cycle (SDLC). As such, the choice of methodology is not one to be taken lightly.

In the emerging days of software development, most developers utilized a waterfall-like method, consisting of a set of phases conducted one at a time. Each phase had to be meticulously planned and executed, seeing as one would not be able to revisit it once it was finished. Once all steps were finished, the product was complete. As software grew more complex, developers started adapting different approaches to developing software, namely agile methodologies. They found that requirements, technology and best practices changed throughout the SDLC, and that such an issue could best be addressed by anticipating changes. Whereas a waterfall methodology could be apt for a specific subset of software development, it would not be applicable for this project. The project consisted of a set of loose requirements, and required the development of several prototypes to explore what the optimal solution would look like. In such cases, opting for an agile approach would be the sensible choice.

With regards to development methodologies, this project is in an interesting situation. Development methodologies are oftentimes applied to projects in which a group of people need to organize their efforts, whereas this project is conducted by a sole developer. Many of the practices and values associated with any specific methodology could provide unnecessary overhead that could slow down development. Being the sole developer of a project brings the benefit of not having to account for the work of others. This was a key factor when choosing the methodology for the project. The author chose to focus on the key values and aspects of each methodology, and what could be applicable and beneficial for a single developer.

Scrum and Kanban are two frameworks often compared when discussing agile methodologies. Scrum emerged with a focus on iteratively delivering increments of code in sprints, a short time period usually spanning one to four weeks [31]. Throughout the sprint, the team conducts short daily meetings, in which one explains what they have done since the last meeting, what is blocking them and how they are planning to solve it. Once a sprint ends, the new functionality is demonstrated to the client, and the team goes through a meeting to discuss how the sprint went and how it could be improved. Sprint tasks are stored in a so-called backlog, a catalog of tasks one must handle at some point during the SDLC. These tasks are usually prioritized based on their importance, and are usually allotted a number corresponding to the expected amount of work it would require to implement. Once a sprint ends, any unfinished tasks are returned to the backlog, and the entire process is repeated.

Kanban, on the other hand, functions differently. Instead of opting for sprints, the Kanban framework aims to create a more continuous flow of releases. This is primarily done through a Kanban board, which consists of several columns detailing what phase a task is in. The columns differ from team to team; Usual columns are "to do", "in progress", "testing" and "done". This approach to tracking tasks is also implemented in the Scrum framework, although under a different name. An interesting aspect of the Kanban board is the Work In Progress (WIP) limit, describing how many tasks one can perform at a time [31]. For instance, each developer can at most have two tasks in the "in progress" column at one time. Once a task is moved to the "done" column and the WIP limit is no longer full, a new task can be chosen from the backlog. This allows for a more reactive development process than the one provided by Scrum. In stark contrast to Scrum, new increments of the project are not delivered in set time periods. Instead, the customer and the development team decides when new deliverables should be produced.

Both Scrum and Kanban provide more ideas and principles than the ones presented in previous paragraphs. However, most of these are more applicable in a team context. Furthermore, Scrum is more prescriptive than Kanban; The framework provides a set of predefined rules and roles that should be used. By opting out of any of the prescribed Scrum concepts, one no longer uses Scrum, but rather

something inspired by Scrum. Kanban is much more adaptive, in that it does not describe any necessary roles or delivery strategies.

One of the major benefits of Scrum is the rigid set of ideas which organizes a team of developers and empowers them to deliver software in a continuous and predictable manner. The perceived trade-off of choosing Scrum over Kanban for this project is the loss of flexibility in terms of pulling in new requirements as they appear, and the general overhead that the prescribed rules and roles would introduce. In a prototype-reliant project such as this one, where requirements are continuously added and re-prioritized after feedback, the added flexibility of Kanban could prove to be beneficial. There is no incentive to provide new functionality in a predictable time-frame, as the only delivery constraint of the project is the final delivery date. As such, the author decided to adopt the Kanban framework.

4.2 Project Management

Project Management tools allows teams to gain a greater understanding of the current state of a project. In order to achieve everything the author wanted, a strict time-frame had to be followed. The use of Trello, a collaborative Kanban board application [32], allowed the author to always stay on top on what had been done, what should be done, and what is being done. Choosing Trello over other alternatives did not have any considerable contributing factors behind it. The author only wanted basic functionality, which most project management tools provides. It was a free alternative, and the author already had previous experience with. As such, it was deemed a good choice. Whereas a larger team would require additional checks and balances, a sole developer really only requires a way to organize their efforts. A spreadsheet or a todo-list could work just as well as any other alternative, but the author chose to utilize a more visually appealing tool.

The Trello board consisted of three columns: To-do, Doing, and Done. It is shown in Figure 4.1. Tasks flowed from left to right, depending on their state of completion. Completed tasks are archived after being in the done column for some time to avoid cluttering the board. Each task is labeled with different colors, to add some additional context to what the task entails.

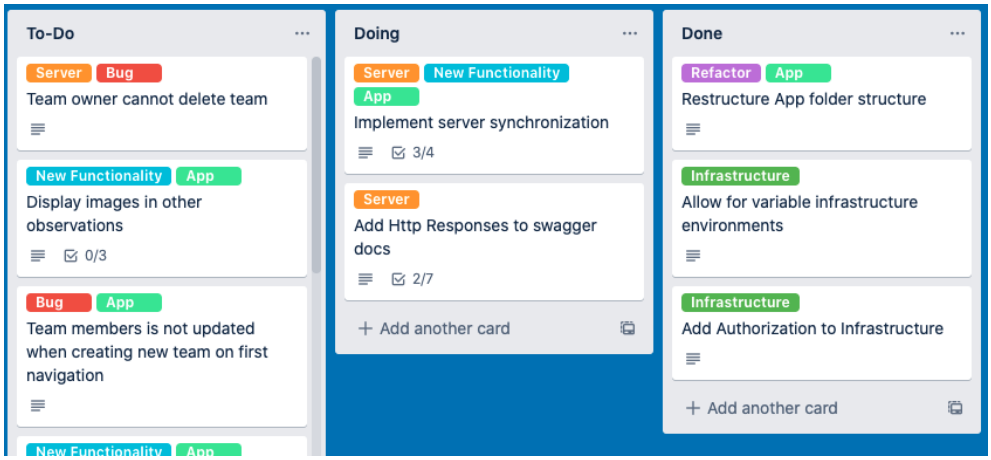

Figure 4.1: Excerpt of Trello board for project.

The author decided to set the WIP limit to three, allowing for some leniency in what should be worked upon. This also allowed for re-prioritizing tasks on the fly, in cases where a certain task would be blocked by another. Larger tasks were sub-divided into to-do lists, where each sub-task would have to be completed before being moved to done. Usually, large tasks were only sub-tasked right before they were implemented. An example of such a task can be found in Figure 4.2.

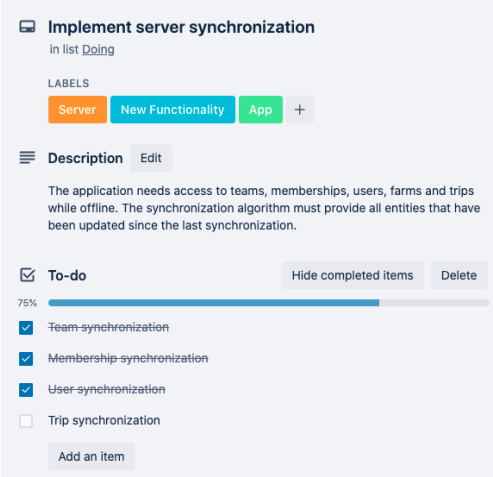

Figure 4.2: A Trello task with sub-tasks.

Beyond the use of Trello, the author also documented what had been done at the end of each day. In many ways, this documentation can be considered to be the equivalent of daily stand-ups for solo developers. These writings all followed the same formula:

- What has been done today?
- What reasons lead to implementing something that way?
- What problems occurred today?
- How can one address the problems? How were they addressed?
- Why did the problem occur in the first place?
- What should be done tomorrow?

By detailing the work at the end of the day, the author was able to compartmentalize the work conducted on any single day. Any potential ideas or solutions were written down to ensure they would not be forgotten. Furthermore, this allowed the author to be more organized with regards to how he worked. Finally, it would drastically simplify the process of writing the actual master thesis, as he would not have to rely upon his memory to recall the development process.

4.3 Software Development Life Cycle: Week-By-Week

The constant in every development week was the meeting with the project supervisor. This meeting marked the end of the current week, and the beginning of a new one. The author would present his findings, prototypes and finished functionality and receive feedback. After these meetings, the author would prioritize the tasks for the following week. If the supervisor provided constructive criticism towards the presented work, this would usually be prioritized the highest.

Every day followed the same general outline. Whenever a new task is chosen, the author would plan the design and implementation through paper sketches and pseudo-code before actually implementing it. Whenever a new prototype was developed, it would be user-tested and improved wherever possible. Any finished or updated tasks would be moved towards the right of the Trello board throughout the day. Finally, once the day ended, the author would document the day and plan out the next one.

Chapter 5

Technical Design: An Overview

The final system consists of several different components. Due to this, the author chose to divide the thesis into several parts. Each part will describe the details relating to the component it covers. This chapter, on the other hand, will provide the reader with a surface-level overview of the final system. Furthermore, it will describe the technologies, tools and architectural patterns shared by all system components.

5.1 System Architecture at a Surface-Level

The system architecture is shown in Figure 5.1. Supervision documentation is performed through a mobile Flutter application that can run on both iOS and Android devices. This part of the system is described in Part II. User Authentication is performed through Auth0, a managed identity and access management solution used for authentication and authorization [33]. The backend of the solution is a RESTful API hosted on Amazon Web Services (AWS). It utilizes a PostgreSQL database. Furthermore, the backend utilizes the same Auth0 instance to authenticate any incoming requests. The system backend is described in detail in Part III, whereas the cloud infrastructure of the entire system is detailed in Part IV. This system can be described as a multi-tier architecture, a pattern that physically separates system components over numerous computing "tiers" [28]. It can also be called a three-tiered architecture; Data persistence, business logic and presentation is physically separated over three separate computational devices.

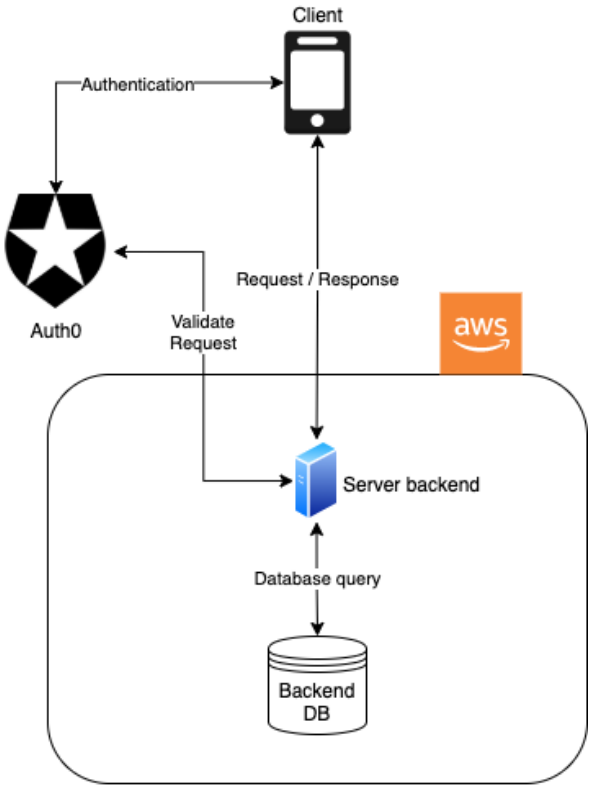

Figure 5.1: A high-level overview of the system architecture.

5.2 Tools & Technologies

Utilizing development tools and technologies properly is essential when designing large systems. Effective usage can drastically increase both productivity and the quality of the end product. This section describes a set of tools used throughout the entirety of the SDLC, and provide some reasoning as to why they were chosen.

5.2.1 Auth0

Both the server and the mobile application requires a way to authenticate users, a very sensitive process one should handle with care. In recent years, products handling authentication and authorization have grown popular. The author believes the usage of such solutions allows for a more secure system, as authentication is a notoriously difficult process to implement securely.

This project utilizes Auth0, an identity management solution providing authentication and authorization as a service [33]. The platform provides the developer with a great deal of customization, and provides a fully configurable user interface for authentication purposes. Furthermore, Auth0 allows for developers to secure

their API's through the use of JSON Web Token tokens, allowing for one to ensure the validity of every request sent to the server. The platform was chosen largely due to its ease of use, extensibility and a generous free tier. Further discussion surrounding the choice can be found in Section 23.2.

5.2.2 Visual Studio Code

The entire system is developed using Visual Studio Code (VSCode), a lightweight Integrated Development Environment (IDE) developed by Microsoft [34]. Whereas many IDE's focuses on a specific programming language or environment, VSCode provides a platform for a plethora of programming languages. This is achieved through the use of VSCode extensions, which adds additional functionality to the IDE. All programming languages utilized throughout the system are fully supported in VSCode, either through extensions or natively.

Having a lightweight and extensible IDE allowed the author to create efficient workflows, and gain an extensive knowledge of the baseline capabilities provided by the IDE. Other IDE's might provide better experiences within specific domains. However, this was not applicable for the technologies this project utilized.

5.2.3 GitHub

GitHub is used for source code versioning and storage [35]. There are no real advantages or disadvantages to choosing GitHub over any other version control system; The author simply prefers it over the other alternatives.

5.2.4 SQL

Structured Query Language (SQL) is a querying language used by many modern relational databases. It allows the developer to query and mutate data stored in relational database tables. Every database present throughout the system uses SQL for querying data.

Part II

Mobile Application

Chapter 6

Application Development Approach

Due to the remote nature of sheep supervision trips, any device larger than a handheld would be cumbersome to work with. With smart phones being a commodity, choosing it as a development platform is ideal. This section provides a brief introduction to mobile platforms before describing the different approaches for mobile development. Finally, it will compare different cross-platform frameworks, and reason as to why Flutter is the chosen solution for this project.

6.1 A Quick Introduction to Mobile

The modern mobile phone, the smart phone, is a mobile device combining impressive cellular and computing capabilities. The formative years of the smartphone saw a market share split between several different operating systems. As the years went on, only two would remain: iOS [36] and Android [37]. At the time of writing, Android devices claim 72.92% of the global market, whereas iOS takes up 26.53% [38].

These numbers are not necessarily representative of the Norwegian market. In fact, October readings show that iOS claims 62.67% of the Norwegian market, whereas Android takes up 37.2% [39]. Producing the application for a single operating system could possibly be easier. This would, however, introduce an unnecessary barrier of entry. In order to keep the application as accessible to the general public as possible, it should function on both operating systems.

6.2 Mobile Application Development

The modern mobile development landscape provides several approaches for developing mobile applications. This section will briefly introduce the different approaches, before finally providing the reasoning behind choosing a cross-platform approach.

6.2.1 Native Development

Native development is the process of developing applications for a specific operating system. Both iOS and Android provides development kits for interacting with the underlying operating systems in an easy, performant and robust way. Furthermore, they allow you to easily build user interfaces resembling other user interfaces already present on the device in question.

Developing mobile applications can be considered a costly affair, both in terms of time and expenses. Opting to go for the native approach can yield in high-performance applications with recognizable user interfaces and arguably a better user experience. However, it comes with the caveat that the application must be developed from scratch for every supported operating system, drastically increasing the amount of required work.

6.2.2 Cross-platform development

Cross-platform frameworks allows one to develop application that can be compiled and ran on several operating systems. How this is achieved differs between frameworks. Some transpiles code to fit the relevant platform, whereas others creates a translation layer for system instructions. Using cross-platform frameworks can allow a developer to drastically reduce time-to-market and costs, due to developers only having to write code once. This logic can also be applied to testing, although some specific platform-testing must be conducted.

The major drawback of the aforementioned frameworks is the performance penalty cross-platform frameworks introduce. Cross-platform frameworks acts as a layer of abstraction between an application and the underlying operating systems, resulting in a longer travel time for calls to the native operating system. Another drawback is platform support. Native applications have full access to the underlying operating system API's, whereas cross-platform frameworks must implement their own API's for calling the API's of the underlying operating system. Frameworks solve this by relying on open-sourced plugins acting as a bridge between the native operating systems and the cross-platform application. However, if the required functionality is not available for the framework, the developer will have to write native implementations for both platforms.

6.2.3 Progressive Web Application

Since the arrival of the Smart phone, mobile devices have surpassed the once dominant computer [40]. This change incentivized companies to adapt their applications to mobile devices, a problem the Progressive Web Application (PWA) attempts to solve. PWA's can be considered to be downloadable websites that can be ran inside a web view, a bare-bones web browser. By utilizing PWA's, developers are able to access a restricted subset of the functionality provided by the underlying operating system. For instance, PWA's can run without internet connectivity, store information on the device and access the camera. If the proposed solution is simple, or possibly already implemented as a web application, converting it to a PWA could be a quick and cost-effective solution to bring it to mobile users.

However, Progressive Web Applications does have cons. Seeing as the application is, in essence, a web application, it must be run within a web view. This additional layer of abstraction introduces some performance penalties. Additionally, a PWA will not be able to make use of the same extent of functionality as native or cross-platform applications.

6.2.4 Hybrid Applications

An alternative to the aforementioned PWA approach is to build a hybrid application. These applications work by displaying content through a web view integrated into the application. In contrast to PWA's, hybrid applications are able to access the underlying platform to a much greater extent through the use of plugins. One of the major benefits of hybrid applications is the freedom of choice with regards to how they choose to develop their solution. Hybrid Frameworks such as Ionic allows you to use web frameworks such as React, Vue, Angular or Svelte [41]. If a web developer wanted to create a mobile application, familiar frameworks could lower the barrier of entry. As with PWA's, the major drawback of hybrid applications are performance. With PWA's, the application simply runs within a web view. However, the hybrid application runs within a web view, which in turn runs inside the hybrid framework run-time.

6.2.5 Choosing an approach

In order to decide what type of application one should choose to develop, there are several factors one must consider. The first, and possibly biggest constraint, is time. This master thesis has a hard deadline. If one were to choose native development, one should expect to spend more time on application development. On the other hand, if you were to choose a cross-platform or web application approach, you would have more time to tweak, perfect and test the application. With the very limited time constraints of this project, a quick development process is crucial. This would also be beneficial with regards to creating functional prototypes.

Another factor to consider is performance requirements. The goal of the application is to provide a simplified way of conducting sheep supervision. Without a pleasant user experience, the benefit of the application could be lost to the target users. The author believes current cross-platform solutions to be performant enough for the purposes of this application. Furthermore, he believes the performance trade-off is worth the reduction in development time. PWA's are not a viable option, as the application requires platform functionality not supported by PWA's. Choosing hybrid applications is a possibility. However, the hybrid approach comes with a larger performance penalty than cross-platform frameworks, with no additional benefits.

The author believes utilizing a cross-platform framework would allow for a much shorter development time, which in turn would free up time for fine-tuning and tweaking to ensure a good user experience. As such, the cross-platform approach was chosen.

6.3 Available cross-platform solutions

The mobile development community has provided several cross-platform frameworks throughout the previous years. This section will describe and discuss the most popular frameworks, before finally explaining why Flutter was chosen.

6.3.1 React Native

React Native [42] has been the front-runner for cross-platform development since its inception in 2015. The framework, developed by Facebook, allows the developer to write their application using JavaScript. UI is defined through React Native's templating language, JSX.

React Native makes use of native UI components; The appearance of components will depend on what operating system the application is running on. The use of the aforementioned native components allows the JSX UI definitions to be compiled into native code, improving performance. Business logic, on the other hand, is ran in a separate thread, and remains as JavaScript code. This leads to a problem: UI and business logic does not run in the same thread. React Native addresses this by creating a bridge between UI and business logic, allowing the application to send messages between the UI and the business logic.

React Native was the most popular cross-platform framework for an extended period of time. Due to this, the surrounding ecosystem is very strong. Support and packages are easily available. Packages are hosted using npmjs.com, a package manager for JavaScript.

6.3.2 Flutter

One of the newer frameworks in the cross-platform ecosystem is Flutter, developed by Google and released in 2017 [43]. The framework is quite unique, as source code is compiled into native machine code. Furthermore, the framework does not make use of native UI components whatsoever, instead opting to draw all components through the use of Skia, a graphics 2d library. With these two unique quirks, many consider Flutter to be more performant than its competitors.

The framework allows the developer to write their application using Dart, a type-safe language written by Google themselves [44]. Although the language is not developed exclusively for use with Flutter, the two are often associated. This can be both a pro or a con, depending on the eye of the beholder. Future development of the language will always account for how it will affect Flutter. However, a niche language like Dart may not see as rapid development as more popular languages.

Flutter has seen an extreme growth in the last two years, surpassing React Native in terms of popularity [45]. The growth has allowed for the community to develop a plethora of third-party packages for the framework, with Google maintaining a list of "core plugins" they choose to officially support themselves. All packages are hosted on pub.dev, a website with the sole purpose of providing Dart packages.

6.3.3 Xamarin

One of the oldest frameworks within the cross-platform space is Xamarin [46]. The framework is built upon the *Mono* project, which aimed to bring the Microsoft *.Net* framework to other operating systems. The development of Mono eventually led to "Mono for Android" and "Xamarin.iOS", implementations of the framework which allowed for developers to create Android or iOS applications with C#. The two frameworks were eventually consolidated into one, simply named Xamarin.

The framework provides cross-platform solutions not only for mobile, but also for web, MacOS and Windows. While the business logic can be entirely shared between all platforms, Xamarin provides several ways in which one can create user interfaces. A subset of Xamarin, named Xamarin.forms, allows one to create platform-adaptable user interfaces.

Packages are provided through nuget.org, the package management solution for the .Net Core framework. Being able to use C# packages drastically increases the functionality provided by Xamarin.

6.4 Choosing a framework

Choosing a framework is possibly the most important decision when developing cross-platform applications. There is usually no clear-cut answer to such a question; The answer entirely depends on the requirements of the application to be developed. In order to gain a better understanding of what framework would be the most beneficial for this project, the following sections will describe a set of properties the author compared the frameworks through.

6.4.1 Maturity and Adoption

Every framework in contention has already underwent several major releases. In terms of maturity, all frameworks have shown to be production-ready.

The framework with the most impressive "resume" of applications is React Native, having been utilized by many major organizations. Their showcase includes applications such as Facebook, Instagram, Skype and Shopify [47]. Airbnb famously chose to abandon the framework in 2018 in a collection of blog posts detailing their reasoning behind their choice [48]. The company listed numerous technical and organizational challenges influencing their decision, citing immaturity of the framework and its surrounding tooling as some of the root technical causes [49]. However, the article also points out that a majority of Airbnb engineers considered their experiences with the framework to be positive. Furthermore, they considered the framework to be approaching maturity [50]. This sentiment appears to be reflected in other companies using React Native. Shopify describes successful implementations of two of their most popular applications, with the framework allowing Shopify to share over 95 % of their total lines of code between Android and iOS [51].

Flutter is another contender with an impressive customer showcase. It has already been adopted by giants like Alibaba, eBay and Google [52]. eBay released a breakdown of their experiences with Flutter, in which they claim that 99 % of the lines of code in their eBay Motors application is shared between the two mobile platforms [53]. Alibaba has also praised Flutter, and even wrote articles about how the framework outperforms React Native [54].

Xamarin also boasts some impressive applications, including the likes of the UPS application and Alaska Airlines [55]. Being the oldest framework of the four, it would be reasonable to assume that it is at the very least as mature as its competitors. Whereas both React Native and Flutter has detailed blog posts about framework adoption, the writer was unable to find any such posts for Xamarin. Microsoft has published several articles about its usage within applications like the ones mentioned above, although one could argue for the inherent bias of such praises. However, seeing as the framework has been relevant for as long as it has been, calling the framework mature cannot be considered to be unreasonable.

Stack Overflow provides an interesting data source through the Stack Overflow Developer Survey. The 2020 survey generated a total of 65 000 replies, and asks questions regarding demographics, technologies and work [56]. All frameworks appears within the "Most loved, Dreaded, and Wanted Other Frameworks, Libraries and Tools" category. The "loved" category describes the percentage of developers using "x" framework and would like to continue doing so, whereas the "dreaded" category describes the percentage of developers using "x", but would like to stop doing so. The "wanted" category describes the percentage of developers interested in trying out "x" framework.

Within the loved category, Flutter is ranked third overall, followed by React Native in 10th and Xamarin in 16th. Within the "dreaded" category, Xamarin ranks 4th, React Native 10th and Flutter 17th. In the "wanted" category, React Native ranks third, Flutter fourth and Xamarin 12th.

Framework	**Loved**	**Dreaded**	**Wanted**
React Native	57.9% (10th)	42.1 % (10th)	14.0 % (3rd)
Flutter	68.8% (3rd)	31.2 % (17th)	10.7 % (4th)
Xamarin	45.4 % (16th)	54.6 % (4th)	4.5 % (12th)

Table 6.1: Stack Overflow Developer Survey 2020 results within the category of "Most loved, dreaded and wanted - other frameworks, libraries and tools" [56].

6.4.2 Performance

In order to provide a good user experience, an application must perform well. Slow frame-rates and unresponsive user interfaces can cause both frustration and confusion, detracting from the perceived usefulness of a solution. Cross-platform frameworks usually introduces a performance penalty, but a well-developed application can still provide enough performance to function well. The field of research surrounding cross-platform framework performance is somewhat sparse. Currently, no scientific papers have compared the three frameworks considered by this thesis. Bjørn-Hansen et al. conducted an experiment in which they compared the performance of cross-platform frameworks, with React Native and Flutter as some of the participants [57]. The study compares the performance of calling different native functionalities on physical Android devices. Flutter and React Native are scored nearly identically with 15 and 16 points, respectively. The Native approach is, in comparison, scored at 24 points out of 30.

Xamarin is the framework with the most research surrounding its performance, as it is the oldest of the considered frameworks. Willocx et al. found that Xamarin applications on iOS devices nearly matched native implementations on high-end devices for UI interactions, whereas lower-end devices usually saw a performance drop [58]. Interestingly, Corbalán et al. showed that computationally intensive tasks performed better on Xamarin than on their native counterparts [59].

Research surrounding the performance of Flutter has not yet been covered by anyone but Bjørn-Hansen et al. However, several companies and developers have created articles about its performance. De Coninck recreated a native application in React Native, Flutter and Xamarin, and compared the CPU usage differences of the implementations [60]. Flutter appeared to outperform the native implementation in both mean and max CPU usage, whereas Xamarin and React Native placed third and fourth, respectively. Additionally, De Coninck described Flutter as the only framework to not show any noticeable frame rate drops during the testing. Similarly, Alibaba compared the efficiency of React Native and a Flutter implementations of their product details page [54]. Flutter outperformed or matched React Native on nearly every test, with the only exception being peak memory usage on low-end iOS devices.

After reviewing the available experiments within the space of performance comparisons of cross-platform frameworks, every considered framework appears to be successful in providing enough performance for a good user experience. Flutter appears to outperform React Native in terms of drawing to the screen, whereas the opposite is true for calling native API's. One could also make a case for Flutter and Xamarin outperforming React Native in terms of performing CPU-intense tasks not dependent on calling native API's, due to the frameworks being compiled to native code instead of being run as a JavaScript process within a virtual machine.

6.4.3 Documentation

In order to utilize a framework properly, documentation is key. Being able to understand why, how or what needs to be done in order to achieve a task allows for quick development times, and usually a better end product. The frameworks considered for this project all provide extensive documentation.

Flutter has an especially impressive amount of documentation. The official documentation website provides guides for getting started, including tailor-made documentation for those who are already familiar with other cross-platform frameworks. Furthermore, it provides tutorials, samples and extensive documentation of the functionality the framework provides. Finally, it provides a set of tips and references one can use to create better applications. Guides for application optimizations, testing, debugging and deployment are all part of the official documentation.

React Native also provides documentation, albeit with the more focused scope of describing what the framework provides and how it can and should be used. Their documentation provides a generic guide for getting started, an overview of the provided components and descriptions of their API's. Every component and API provides example usages.

Xamarin documentation covers a much broader scope than the aforementioned frameworks. Their documentation provides a detailed, albeit generic, guide for

getting started. Both components and API's are detailed within the same collection, making the process of navigating the options somewhat more difficult. Examples are not always present within a specific component or API description, oftentimes being explained in separate code samples or in the "getting started" guide. Overall, the Xamarin documentation proved to be more difficult to navigate than the ones of Flutter and React Native.

6.4.4 Support for functional requirements

In order for a framework to be a viable option, it should be able to fulfill the functional requirements of the project. In order to assess this, the author read the documentation provided by every framework and explored their respective package managers. The following functionality was what the writer deemed to be absolutely mandatory for a framework to be deemed viable:

- Support for maps. Should support custom tile providers and offline tile storage.
- A database solution.
- Location tracking, both when the application is active and inactive.
- Camera support.

Every framework appeared to support the required functionality, either provided directly by the framework or through third-party packages. In general, the packages provided by React Native had the overall best quality in terms of documentation and active maintenance. Most packages had reached maturity, and had an active community surrounding it. Flutter packages were all actively maintained, although with a lower quality of documentation. This is to be expected, seeing as the framework is newer than React Native. Xamarin proved to be a mixed bag. Some packages appeared to be well maintained, with ample documentation. Deprecated and migrated packages proved to be a reoccurring theme, making it hard to understand what packages were up-to-date.

In large, no framework showed a large advantage in terms of support for the functional requirements, besides the fact that Xamarin would require the developer to do some additional development in order to display offline maps.

6.4.5 Tooling

One of the major benefits of cross-platform development is the drastically reduced time-to-market. Good tooling enables developers to write code more effectively. Code formatting, code suggestions, hot reloading and a well-functioning IDE can drastically accelerate development.

React Native provides an extensive set of tooling, with the most important one possibly being the hot reload functionality. Hot reloading allows a developer to

automatically inject changed source code into a running application while retaining the current state of the application, all without having to rebuild and re-run anything. This drastically increases development downtime and improves overall efficiency. The framework is supported in several IDE's. By having framework support, the developer will gain access to code suggestions, linting and built-in tooltips. Debugging can be performed through Google Chrome's developer tools or through a standalone application.

Flutter also provides excellent tooling. The Flutter plugin [61], available for Visual Studio Code and Android Studio, introduces several helpful features: Code auto-formatting, code suggestions, code linting and built-in tooltips. Furthermore, Flutter is the only framework to support both hot reloading and hot restarting. Whereas hot reloading retains the state of the application, a hot restart injects changes into the running application whilst also resetting the state. Performing such an action in React Native or Xamarin would have the developer stopping, rebuilding and restarting the application. Furthermore, the framework provides excellent debugging tools, which can be used to debug code or inspect a number of things. At the time of writing, one can inspect memory usage, performance, network activity, app size and widget states.

Xamarin allows for developers to develop in Visual Studio or IntelliJ Rider, with only the former providing a free tier. The free tier lacks several important features with regards to Xamarin tooling, with the biggest losses being the Xamarin Profiler, Xamarin Inspector and code coverage. Besides the aforementioned constraints, Visual Studio provides code formatting, static code analysis and tooltips. Like the other frameworks, Xamarin provides hot reloading capabilities, albeit with less power; Only certain changes to the application user interface is supported.

6.4.6 Choosing Flutter

Choosing a cross-platform framework is an important decision, and not one to be taken lightly. In this project, the framework was chosen through a process of elimination.

Xamarin is the oldest of the frameworks, and has been used by several successful companies since its inception. However, continuous and major changes within the structure of Xamarin is considered to be somewhat worrying, introducing fears with regards to the possible longevity of the developed solution. Furthermore, the placement of the framework within the "dreaded" category in Stack Overflow's Developer Survey is worrisome, and possibly a reflection of the developer experience. Finally, the lacking hot reload functionality is considered to be a major drawback, as it will decrease developer productivity. Due to the aforementioned factors, the writer decided to not choose Xamarin.

Flutter and React Native both appear to be proven frameworks with widespread adoption. The writer considers React Native to be slightly more mature, mostly due to it's age and the amount of high-quality third-party packages within its ecosystem. However, Flutter appears to have many packages of the same quality, and possibly better tooling than its competitor. Furthermore, with the rapid popularity of Flutter, the amount of high-quality packages are certain to increase. Flutter also appears to have a slight advantage in terms of UI performance and application smoothness, causing the writer to lean towards the framework. In the end, the choice came down to the development experience. After experimenting with both frameworks, the author ended up preferring Flutter. The choice is mainly affected by a perceived productivity increase, as he was able to create user interfaces more easily using Flutter. With the planned project being heavily reliant on prototypes, choosing the framework with the highest productivity boost is sensible.

6.5 Application Development Technologies

This section will provide brief descriptions of the technology used to enable or simplify the development of the mobile application.

6.5.1 Android Studio

Android Studio is an Integrated Development Environment (IDE) focused on Android development, and provides an extensive set of tools that simplifies the process of developing Android applications [62]. In the case of this project, Android Studio is only used for creating and managing Android Emulators, which allows developers to run Android applications on computers.

6.5.2 XCode

In the same vein as Android Studio, XCode is an IDE provided by Apple for developing applications for Apple devices [63]. This application mainly utilizes XCode for iOS simulator access. Furthermore, XCode is utilized to enable iOS-specific settings only available through XCode.

6.5.3 Flutter and Dart Plugins for VSCode

The entirety of the project has been exclusively developed through the use of Visual Studio Code (VSCode), a lightweight IDE fit for several development processes. Support for Dart and the Flutter framework is not available by default. Instead, it is provided through two VSCode plugins, named "Flutter" [61] and "Dart" [64]. By utilizing the two plugins, the developer is able to perform every action required for Flutter development inside VSCode.

Chapter 7

Application Overview

This chapter provides the reader with a full overview of the user interfaces available throughout the mobile application. Every section is dedicated to a specific domain within the application, and will describe how the user can interact with it, why certain choices were made, and any associated problems that required solving. The application features a dark theme and a light theme, conforming to settings in the underlying operating system. This overview will only provide screenshots of the light theme.

7.1 Authentication

The Application landing page only provides the user with a single option: signing in. By pressing the sign-in button, the application launches an in-app browser, allowing the user to sign in, sign up, or reset their password. Mobile development best practices states that browser-based login authentication solutions are preferred [65]. Sign-up, login and forgot password-functionality were all initially intended to be embedded into the application. However, this was later changed to conform to RFC 8252.

The user interface for authentication is provided by Auth0, an Identity and Access Management solution. Figures 7.1 and 7.2 displays the views for logging in and resetting password, respectively. The UI for signing in is omitted, as it is nearly identical to the user interface used for signing up.

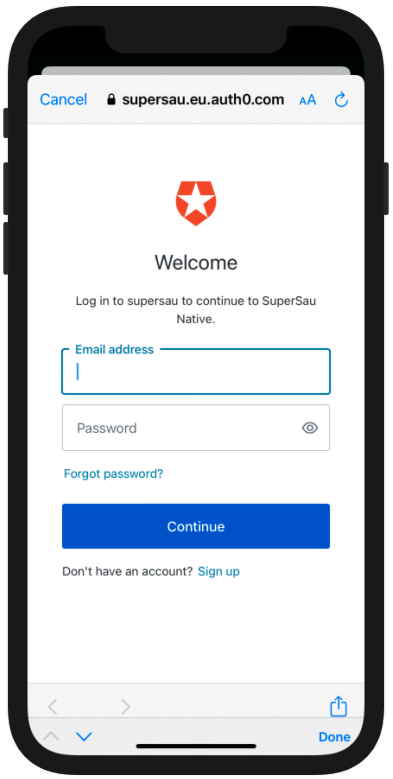

Figure 7.1: Signing in.

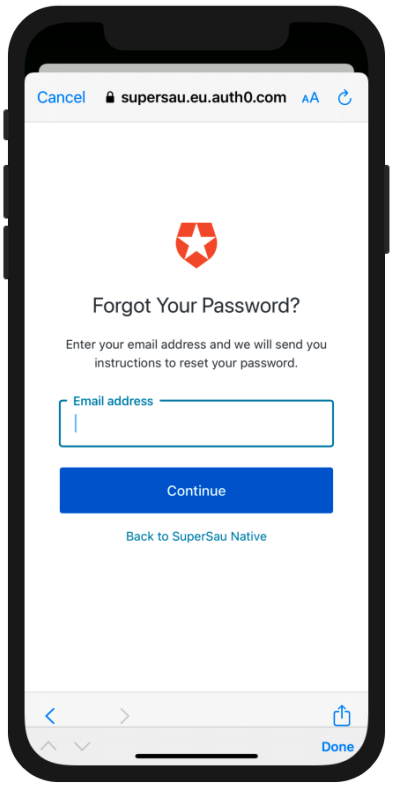

Figure 7.2: Reset password.

7.2 Detail Registration

Once a user has created an account, the application prompts the user for additional details. The only requested user information is the first and last name of the user, allowing one to easily identity users within teams. The user interface simply consists of input fields for the aforementioned values and a confirmation button. It is shown in Figure 7.3. Once the user submits their details, they are redirected to the Home View.

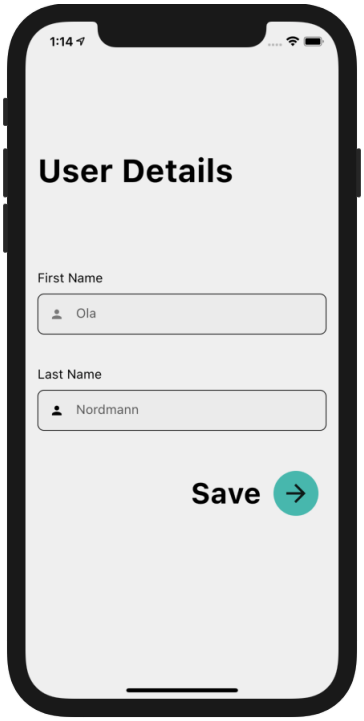

Figure 7.3: Registering user details.

7.3 Home

Logged-in users are immediately navigated to the home screen, which provides access to every other view in the application. The layout of the home view conforms to the state of the application. When everything is normal, the home screen simply provides navigation to other views, as shown in Figure 7.4. In cases where the application is closed during an ongoing trip, an additional UI component is appended atop the navigation options, allowing the user to either continue the trip or abort it. This is shown in Figure 7.5. The ability to start a new trip is completely removed until the ongoing trip has been addressed. Completed trips must be synchronized with the server state. In order to remind the user of this task, another item is shown on the home screen whenever the user has unsent trips, as shown in Figure 7.6. The only way of removing the "unsent trips" component is by uploading the trips to the server, ensuring synchronization between the local state and the server state. Alternatively, the user can choose to delete the trips locally, in which case the upload option will disappear.

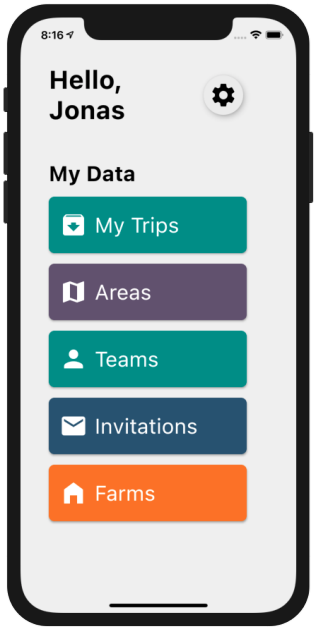

Figure 7.4: Normal home.

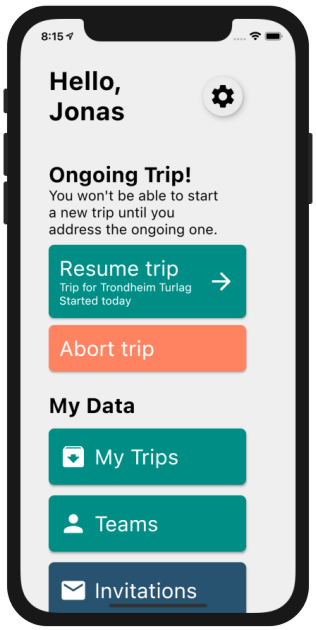

Figure 7.5: Home with ongoing trip.

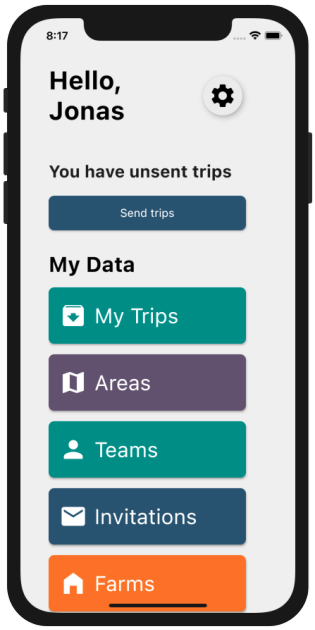

Figure 7.6: Home with unsent trips.

7.4 Farms

When conducting supervision trips, the farms a herd of sheep belongs to should be registered whenever possible. In order to enable a single source of truth for farms, and to reduce the chance of human error, all farms are stored on the database. The Farm view is used for downloading these farms locally to a device, enabling sheep observers to use them during supervision without a need for network connectivity. If no farms have been downloaded, the view simply describes the implications it will have when conducting supervision trips. This is shown in Figure 7.7.

Figure 7.7: No farms downloaded.

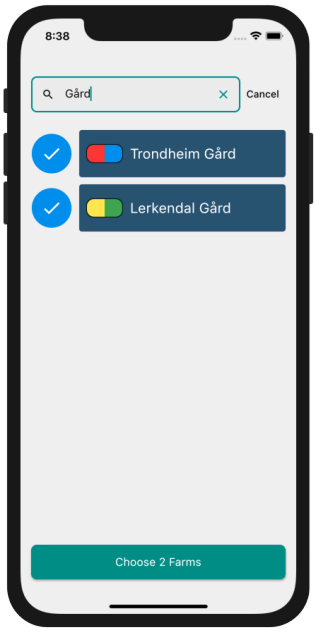

Figure 7.8: Downloading farms.

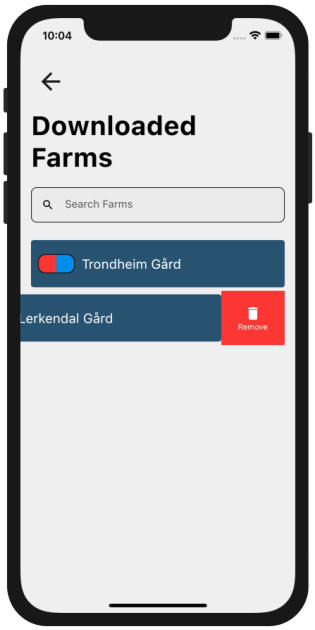

Figure 7.9: Downloaded farms.

By tapping the search input, the user is taken to a search view. Whenever the query string changes, the server responds with a list of farms matching the query. Tapping any query item will result in it being selected for download. Selected farms are indicated through the use of a check symbol within the circle on the left-hand side of the item. In Figure 7.8, both farms are selected and ready to be downloaded. Once the user presses the confirmation button at the bottom of the screen, the farms are stored locally on the device and the user is navigated back to the initial farm view. The farm view automatically updates to display downloaded farms. A locally stored farm can be removed by swiping an item towards the left. The swiping motion will enable a hidden delete button, a common deletion method occurring throughout the application. Figure 7.9 shows the UI of downloaded farms and the appearance of the slide-to-delete action.

7.5 My Invites

The invite view displays a list of pending invites to sheep supervision teams. Each invite can either be accepted or declined. If the invite is accepted, the team will be downloaded and made available throughout the application. By declining, the invite is simply removed from the application. The invite view is displayed in Figure 7.10. If one wishes to send an invite, the inviter must be an administrator of a supervision team.

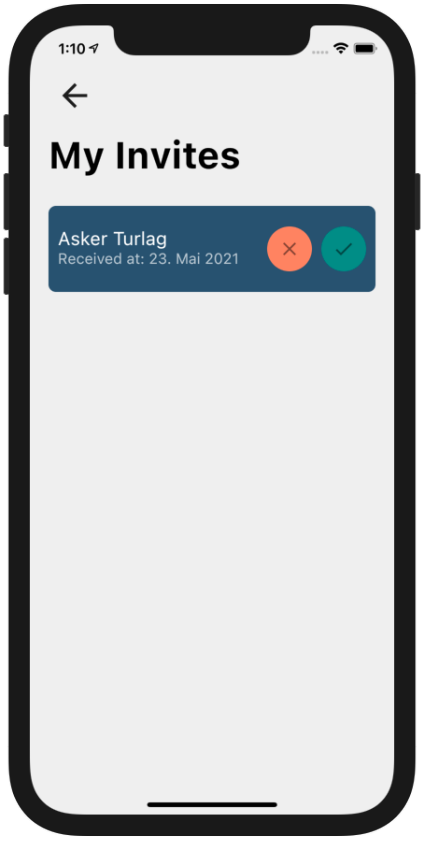

Figure 7.10: My invites.

7.6 Teams

This section describes all views relating to the concept of supervision teams. When navigating from the home view, the user will be met with a list of the teams they are a member of. This is displayed in Figure 7.11. If the user is not a member of any teams, a placeholder text will appear instead, as shown in Figure 7.12. By pressing the plus sign in the top-right corner, the user can create a new team. They are then navigated to a new view, in which one can provide a team name and an optional description. If the user chooses to create the team, they will

be automatically navigated to a detailed view of the team information and the available team actions. The team creation view is shown in Figure 7.13.

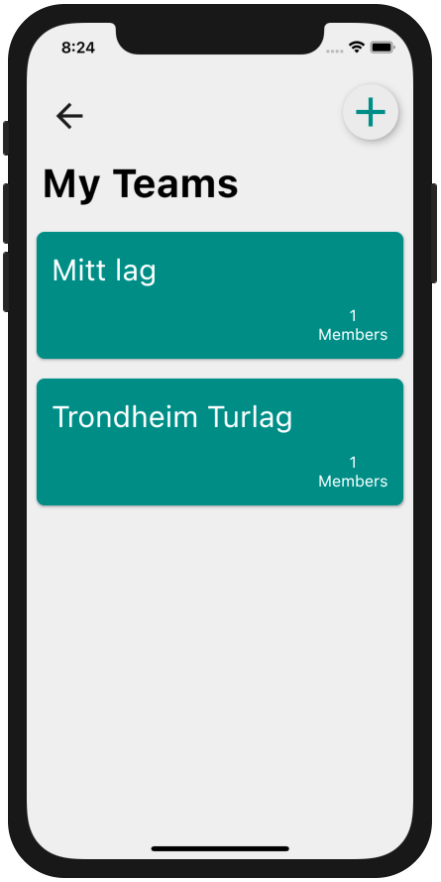

Figure 7.11: Joined teams.

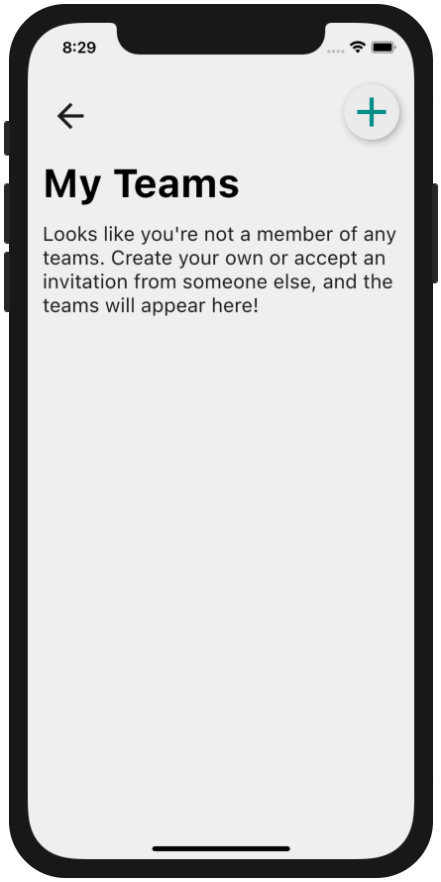

Figure 7.12: No teams joined.

By pressing one of the teams in the team list, the user is navigated to a view providing additional details about the team and a set of actions one can perform within the team. The available actions differ based on whether the user is a team administrator or a regular member. A regular team member can only view the team description, the team members and the trips performed by the team. They may also leave the team. Administrators, on the other hand, are allowed to manage invites, transfer administrative privileges to another team user, remove members, leave the team or delete the team. The administrator team view is shown in Figure 7.14, whereas the administrative actions are shown in Figure 7.15.

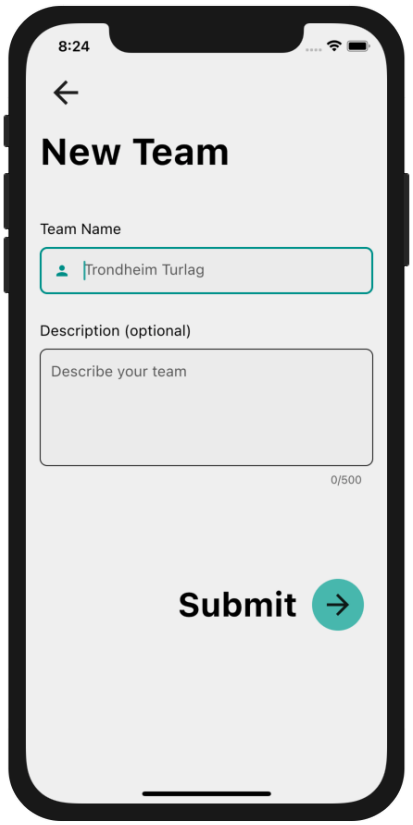

Figure 7.13: New team.

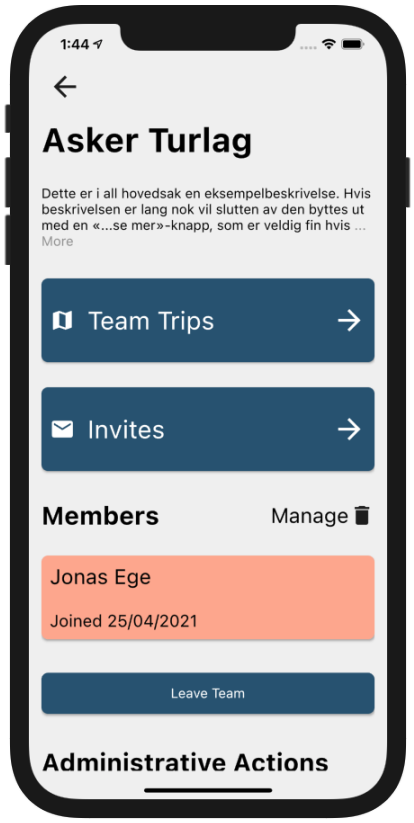

Figure 7.14: Team details.

The "Team Trips" button allows the user to view trips performed within a team. This view is identical to the one discussed in Section 7.7, albeit with a different set of trips. Trips cannot be deleted within "Team Trips". Deleting trips can only be achieved from "My Trips", and can only be done by the original performer.

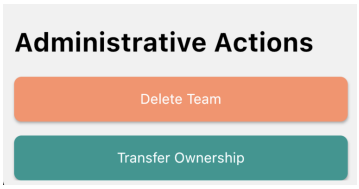

Figure 7.15: Administrative team actions.

Attempting to leave or delete a team results in the application prompting the user to confirm or cancel their choice. The prompt adapts itself to match the underlying operating system, and is shown in Figure 7.16. Administrators can only leave a team after transferring administrative privileges to another member. The team can be deleted if the administrator is the only remaining member.

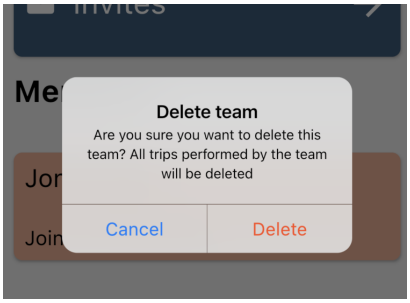

Figure 7.16: Team deletion prompt.

Team administrators can administer team invites by pressing the "Invites" button on the detailed team view. Pressing the button brings the user to a view displaying pending invites, as shown in Figure 7.15. The user can delete multiple invites at a time by using selection functionality akin to the one used when downloading farms. New members can be invited by tapping the "New Invite" button in the top-right corner, and will bring the user to the "New Invite" view. The view only allows the user to provide an email address and submit it, as shown in Figure 7.18. Email validation is performed automatically to ensure data validity. Once the email is submitted, the user is redirected back to the team invite overview.

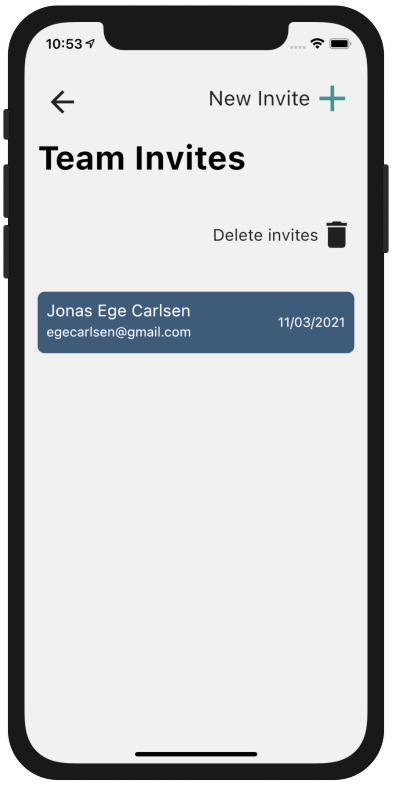

Figure 7.17: Team invites.

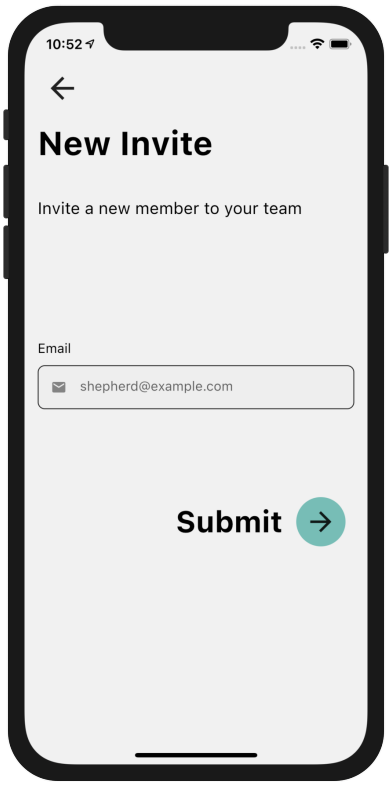

Figure 7.18: New invite.

Transferring team ownership can be achieved by tapping the "Transfer Ownership" button, as shown in Figure 7.15. By tapping the button, the user is navigated to a new view listing all team members excluding the team administrator. This view is shown in Figure 7.19. If the team only consists of the team owner, the ownership transfer view will explain why no items are shown. By tapping a membership, the user will be prompted with a confirmation dialog. By accepting, the ownership of the team is transferred to the selected member. This action is effective immediately; The previous team owner will no longer have access to administrative actions.

Removing members from a team can only be performed by the team administrator, and can be achieved by tapping the "Manage" button shown in Figure 7.14. The button will navigate the administrator to a new view displaying all removable team members, as shown in Figure 7.20. By tapping a membership, it will be marked as selected. Once a membership is selected, the bottom of the screen will display a deletion button. Tapping the button will remove the members from the team and navigate the administrator back to the detailed team view.

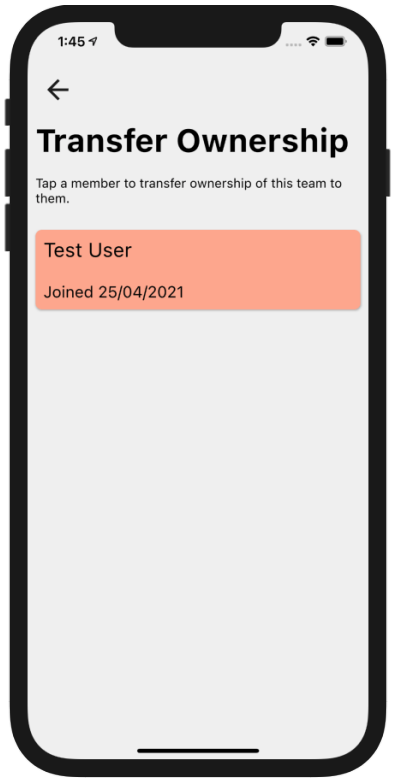

Figure 7.19: Transferring team ownership.

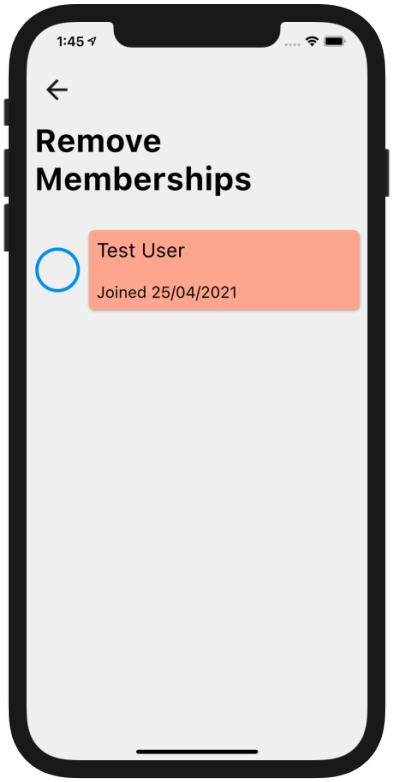

Figure 7.20: Removing team members.

7.7 My Trips

This view displays the trips conducted by the logged-in user, and is shown in Figure 7.21. Each trip item displays the date of the trip, and at what times it started and ended. Furthermore, the icon on the left denotes whether the trip has been submitted to the server or not. Clicking on a specific trip navigates the user to a new view, in which one can view the trip observations.

Figure 7.21: My Trips.

Trip observations are separated by type, as shown in Figures 7.22 and 7.23. This separation is consistent throughout the entirety of the application. In the case of the detailed trip view, they are separated into different tabs, providing a clear overview of what observations have been made. A sheep observation lists the time of which the observation was made, how many sheep were observed and their sheep colors. If the observation occurred reasonably close to the sheep observer, the observation may also include tie colors and farm colors. An observation within the "other" category lists the type of observation, a description of the observation and an optional list of images and image descriptions. Locally stored images are loaded from the file storage and displayed immediately, whereas remotely stored images can be loaded by pressing a button. Observations can only be deleted before they are submitted to the server. Once submitted, they are considered to be final.

Figure 7.22: Sheep Observations.

Figure 7.23: Other Observations.

7.8 Offline Areas

In order to perform a supervision trip, the user must first download a map selection. A stable internet connection is not a given during supervision trips; Having a locally stored map would allow trips to be performed without any need for internet connectivity. The "Offline Areas" view lists every downloaded map portion. Tapping a list item displays a list of map options in a bottom sheet. Each map can be deleted, viewed or used for starting a trip. If the user chooses to view an offline map, they are navigated to a view displaying the downloaded map selection. By choosing to start a new trip, the bottom sheet prompts the user to choose a team for the upcoming trip. Once a team has been chosen, the user is navigated to a new view and the trip is started. This process is displayed in Figures 7.24, 7.25 and 7.26.

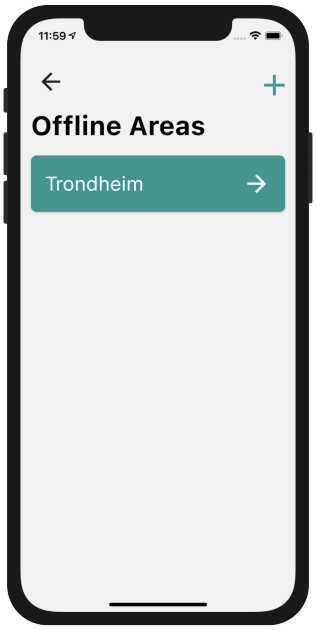

Figure 7.24: Offline areas.

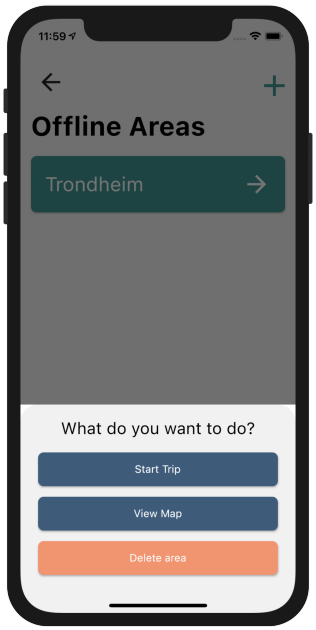

Figure 7.25: Offline area options.

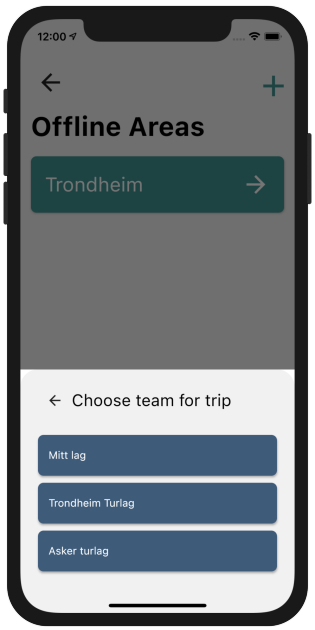

Figure 7.26: Choosing a team for a trip.

The "Offline Areas" view is only available to users when a trip is not currently in progress. If an ongoing trip is present, the only way to navigate to "Offline Areas" again is by finishing or aborting the ongoing trip.

In the top-right corner of the view, the user can tap the "plus" button to create a new offline area. This will redirect the user to a new view, in which a map area can be selected and downloaded. The map supports the same types of interaction as most other mobile map applications: drag to scroll and pinch to zoom in or out.

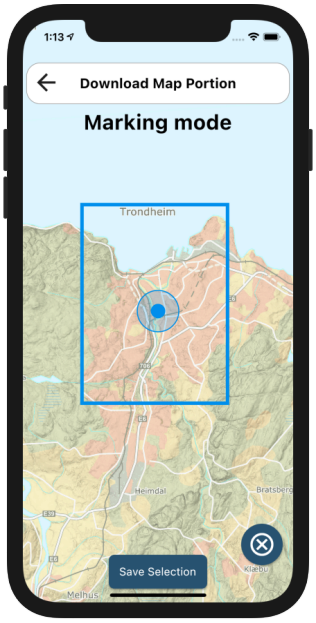

Figure 7.27: Specifying map selection.

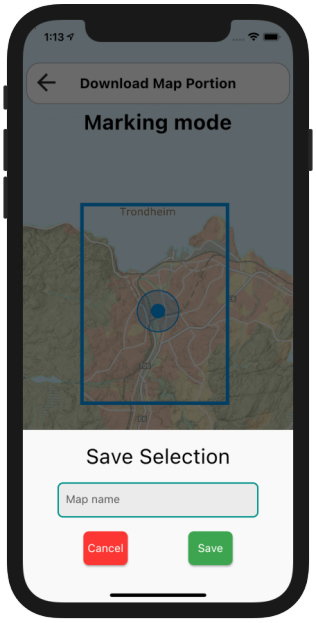

Figure 7.28: Specifying map name.

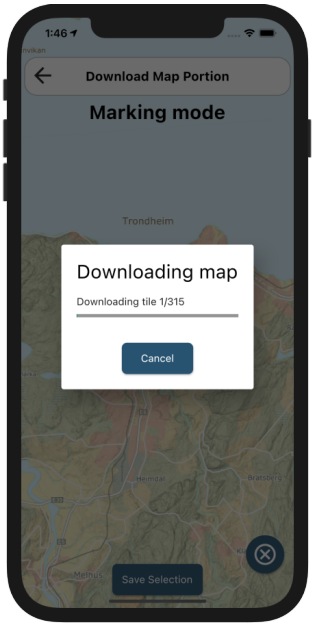

Figure 7.29: Map download indicator.

A map selection can be made by pressing the "plus" sign in the bottom-right corner. Pressing the button enables marking mode, and is communicated to the user through text. Marking mode locks the map in the current position, and allows for the user to mark a portion of the map by dragging from one screen position to another. This is shown in Figure 7.27. Once a selection has been made, the user can choose to save their selection by pressing the "save" button on the bottom of the view. By pressing the button, a bottom sheet is shown, prompting the user to enter a name for the map and to either download it or cancel the map creation process. This is shown in Figure 7.28. If saved, the application will display a progress indicator until the map has been downloaded, as shown in Figure 7.29. Pressing the cancel button will cancel the entire download process and remove any map tiles associated to this specific download. Once a map has been downloaded, the progress indicator disappears, and the newly downloaded area will be listed once the user navigates back to the "Offline Areas" view.

7.9 Trip

The trip view is the main hub during supervision trips. Once a trip has started, the user is only able to exit the trip by finishing it. The exception to this rule is whenever the application is closed, either by the user or the operating system. In such cases, the user is presented with the option of resuming or aborting the trip when the application is opened again. The trip view is displayed in Figure 7.30.

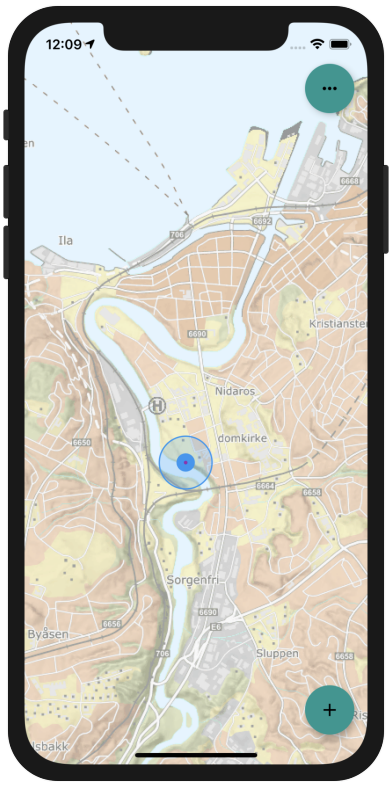

Figure 7.30: Trip UI.

During a trip, the application continuously tracks the user location in order to provide a list of coordinates covered throughout the trip. Whenever a user observes something noteworthy, it can be selected by tapping the "plus" button in the bottom right of the view. This enables a draggable selection cross-hair. When the user lets go of the cross-hair, a selection is made. The application then displays a bottom sheet detailing the distance between the selected point and the person conducting the supervision trip. Furthermore, it prompts the user to specify what type of observation the selection is. This is shown in Figure 7.31. Once a selection is made, the user is navigated to the view representing the observation type. Currently, the application supports sheep observations and "other" observations.

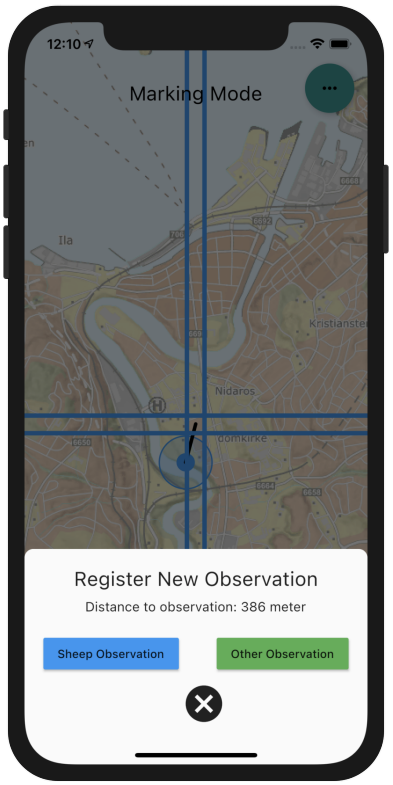

Figure 7.31: Observation type prompt.

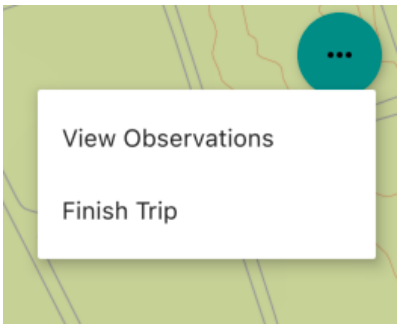

Figure 7.32: Trip pop-up options.

The user is represented as a blue dot on the map, and updates itself in real-time. Whenever the user moves, a purple line is drawn to indicate previous user locations.

Trip actions can be accessed by pressing the button in the top-right corner, which will display the pop-up menu shown in Figure 7.32. This menu provides the user with the option of either viewing previous trip observations or ending the trip. The trip observations view is the same one as described in Section 7.7, and will not be discussed further. If the user chooses to end the trip, the application will navigate back to the Home view.

7.10 Sheep Observations

The primary goal of the application is to simplify and improve the process of registering sheep. As such, the sheep observation view has to be as efficient as possible. The sheep observation view only requires the observer to register the total number of sheep and the number of sheep divided by wool color. In fact, if the observation is more than 30 to 50 meters away, the user is unable to register anything else. The project supervisor specifies that registering sheep details from

distances further than 30 to 50 meters introduces a significant margin of error. As such, a choice to simply disallow additional information on long distances was made. The difference between the two user interfaces are shown in Figures 7.33 and 7.34.

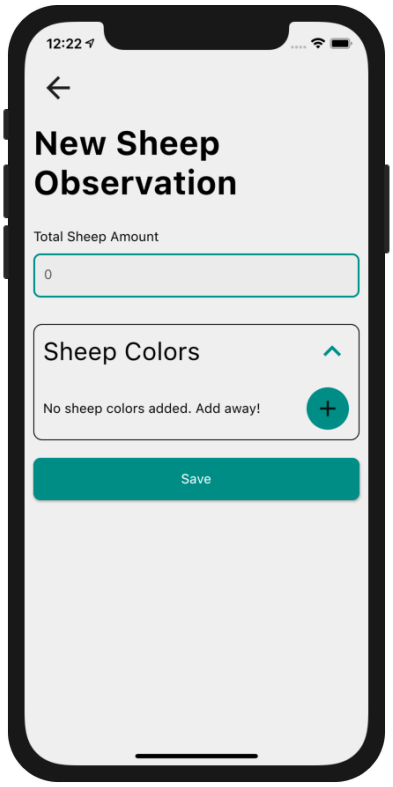

Figure 7.33: Sheep observation details if distance to observation is more than 30 meters.

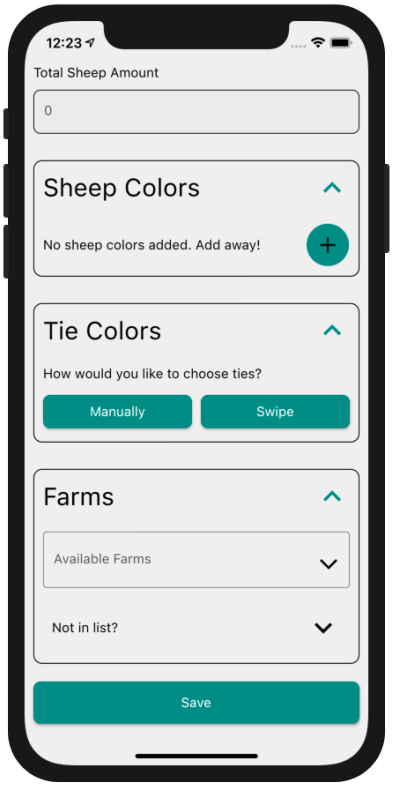

Figure 7.34: Sheep observation details if distance to observation is less than 30 meters.

If the distance to the observation is small enough to observe details, the observer should strive to do so. Each optional detail section is capable of expanding or collapsing to reduce the amount of UI clutter as the registration process goes on. Once a section is finished, it can be collapsed and hidden from the user interface. This can be observed in Figure 7.35, where both the sheep color and tie color sections have been collapsed.

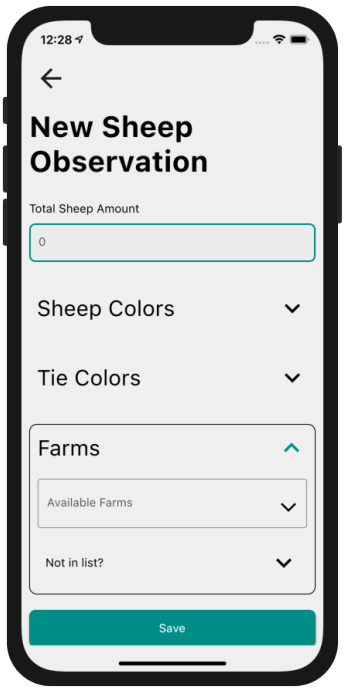

Figure 7.35: Collapsed sheep and tie color sections.

The process of registering a sheep color requires two inputs: a color and an amount. Sheep colors are predefined, and selected from a drop-down list. Every color value throughout the application provides a textual counterpart to accommodate for color-blindness. The amount is a simple integer input field. No sheep color can appear more than once, and the amount must be greater than zero. The application notifies the user of invalid values, as shown in Figure 7.36. Once the list reaches the same length as there are values in the drop-down menu, the ability to add new rows are removed. Figure 7.37 shows the button used for adding more values.

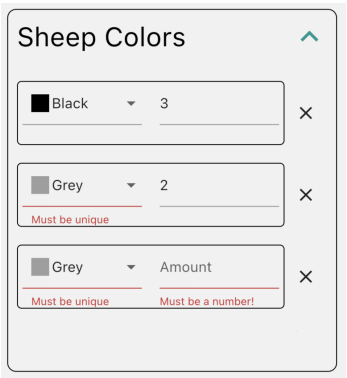

Figure 7.36: Sheep colors with errors.

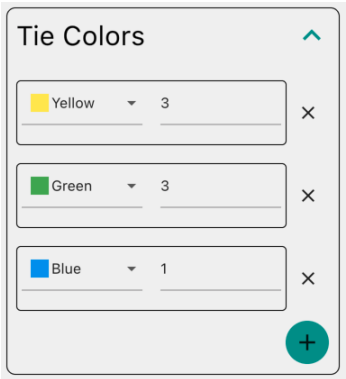

Figure 7.37: Tie color section.

The process of registering sheep ties is, in principle, equal to registering sheep colors. Sheep ties are quite small, and oftentimes hard to keep track of. Furthermore, sheep are not guaranteed to stay still while being observed. Attempting to register ties without constantly looking at the herd can be an error-prone process. Due to this, the application allows the observer to register tie details by swiping. Sheep can have four different tie colors. By assigning one color to each screen edge, the observer can simply swipe to the screen edge corresponding to the tie color. The approach to tie swiping is shown in Figure 7.38.

When swiping, the application also provides a text-to-speech functionality specifying how many times a direction has been swiped. If one were to swipe left for green, the application would respond "Green one". If the next swipe also corresponds to green, the application would respond "two", omitting the color and only mentioning the count. This expedites the swiping process, as the user already knows what direction they are swiping. If a new swipe has a different direction than the previous, both the color and count would be spoken again. By combining swiping and text-to-speech, the application simplifies the process of registering ties without looking at the screen. The text-to-speech is able to confirm what color and amount has already been registered. In order to provide additional confirmation to the user, the amount of times a direction has been swiped is also shown in the UI.

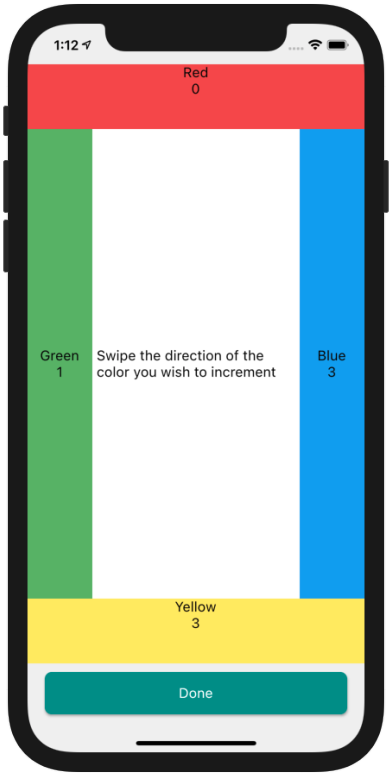

Figure 7.38: Tie swipe UI.

The final optional value is farms. As described in Section 7.4, farms are intended to be downloaded before a trip has begun. Only allowing one to specify downloaded farms would be very naive; The chance of encountering sheep belonging to unknown farms is always present. As such, this section allows the user to define unknown farms by adding the two colors that identifies the farm. The farm section is depicted in Figure 7.39.

Downloaded farms appear in a drop-down list. By pressing an item in the list, they are added to a list of farm observations. This same list houses all unknown farms. When an unknown farm is defined, it is simply added to the list like a downloaded farm, only with a placeholder name of "Unknown".

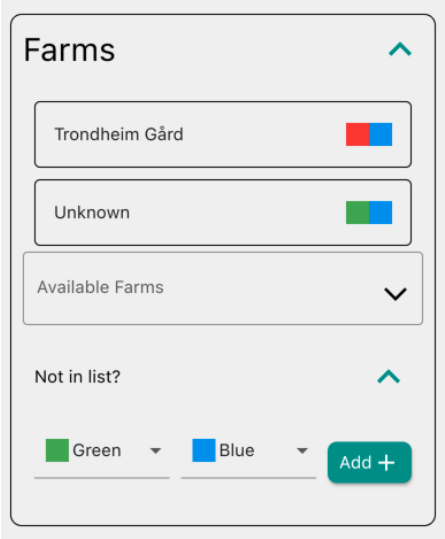

Figure 7.39: Farm section with expanded "missing farm" field.

The observer is only able to save the observation if all details are valid. If not, the invalid fields will provide an error message, describing why they are invalid. Once everything is valid and the observation is saved, the application navigates the user back to the trip view.

7.11 Other Observations

Whereas sheep observations have very specific requirements, "other" observations do not. There are a number of things observers might find noteworthy to register during a supervision trip. Predator sightings and dead sheep, for instance, are two important events one should register.

The "other observation" view requires the observer to register what type of observation is being made. Furthermore, it allows the user to provide a description of what has been observed. It also allows for the observer to append a number of images of the observation. Viewing a full-sized image can be achieved by tapping one of the observation image items. Each picture allows for an optional description of the image. A picture of the user interface is provided in Figure 7.40, whereas the view for taking pictures is shown in Figure 7.41. When taking photos, the user is able to enable and disable flash, as well as swap between the front- and back-facing cameras.

As with the sheep observations view, all fields must be valid before submitting. Once the user has successfully submitted the observation, the application navigates back to the trip view.

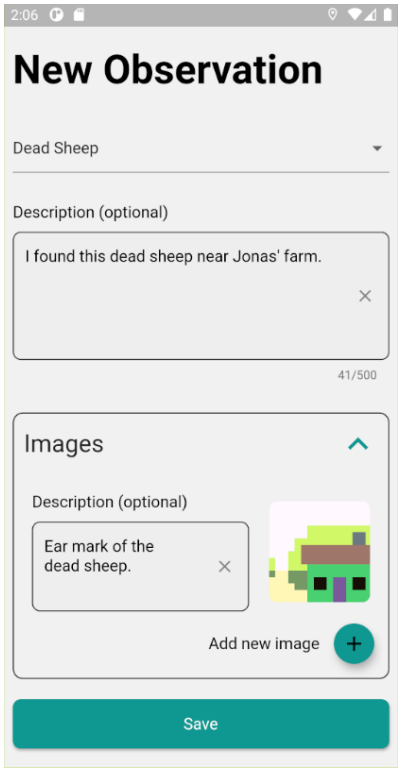

Figure 7.40: Registering an "other" observation

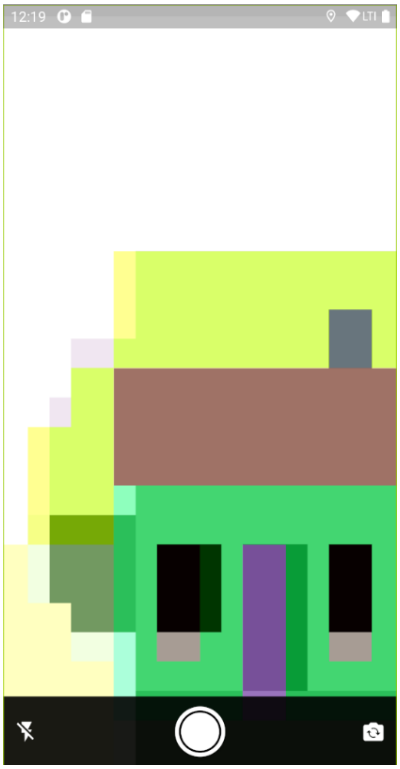

Figure 7.41: Camera preview. Camera is emulated.

7.12 User details and Settings

The User view can be accessed by pressing the gear icon at the top-right corner of the Home view, and provides access to user details and application settings.

User information is displayed at the top of the view. By tapping the edit icon, the user is able to change their first and last name. All input fields must be valid in order for changes to be stored. The differences between the normal state and the editing state are shown in Figures 7.42 and 7.43.

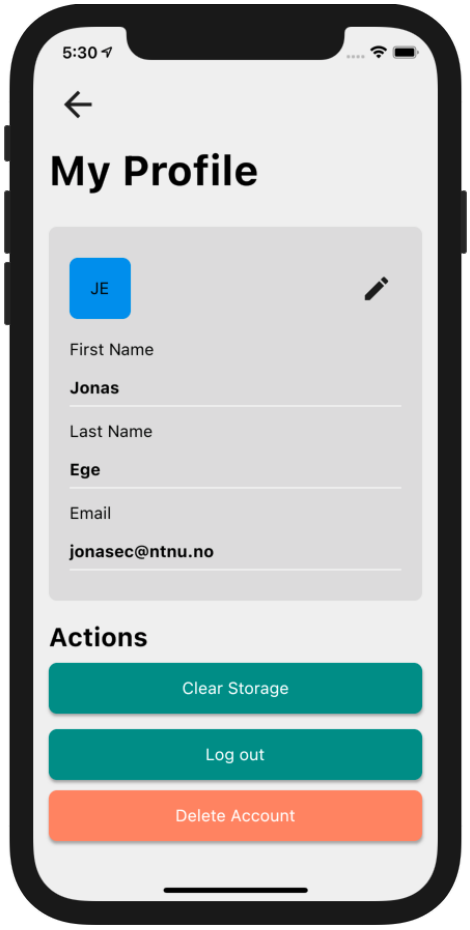

Figure 7.42: User profile and actions

Figure 7.43: Editing user details.

Available user actions are listed at the bottom of the view, and currently consists of three choices. The "Clear Storage" button allows the user to delete all data stored on the device, with the exception of the user session and downloaded maps.

Logging out of the application clears the application data and navigates the user

to the authentication view. If the user chooses to log back in later-on, user data will be fetched from the server. The only data that cannot be retrieved from the server are pictures and downloaded map areas. It is important to note that data deletion only occurs when the user chooses to log out. If the user is logged out due to an expired session, all data remains intact.

Deleting the currently logged in user account can be achieved by pressing the "Delete Account" button. This will delete all application data. Furthermore, the server will modify any database table containing personally identifiable information to comply with GDPR, before marking the account as deleted. Finally, the identity provider account is deleted.

Chapter 8

Flutter: Concepts and Packages

In order to gain a better understanding of certain choices made throughout the development process, it is important to understand the core principles and concepts Flutter is built upon. This section is dedicated to providing the reader with succinct explanations of central concepts that can provide additional context to the rest of this part. Furthermore, it will describe important packages utilized by the project.

8.1 Widgets

"Everything is a widget" is a common saying within the Flutter community. Where similar frameworks would instruct the user to add styling through cascading style sheets, Flutter tells the user to wrap the target widget in another widget.

Code listing 8.1.1: Flutter widget example.

```
class Example extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Center(
      child: SizedBox(
        height: 100,
        width: 100,
        child: Container(
          padding: const EdgeInsets.all(5),
          decoration: BoxDecoration(
            borderRadius: BorderRadius.circular(5),
            color: Colors.blue,
          ),
          child: Center(
            child: Text("Hello"),
          ),
        ),
      ),
    );
  }
}
```

Listing 8.1.1 describes this approach well. The build function returns a padded blue box with circular corners and the text "Hello" centered inside of it. At the top-level, the *Center* widget ensures that the output will be centered in the middle of the screen. Following this, *SizedBox* ensures that the created widget has a height and width of exactly 100 units. A *Container* is a convenience widget, and allows for the developer to add common properties, defined as widgets, to itself. In this case, it allows the developers to pad the child of the container with 5 units on all sides. Furthermore, it allows the developer to specify container decorations; The background color should be blue, and the borders of the container should have rounded corners. Finally, the text inside the widget is defined by a *Text* widget, which itself is centered within the container by wrapping it with yet another *Center* widget. The final result of the aforementioned code can be seen in Figure 8.1.

Figure 8.1: The resulting user interface of the code listing in Listing 8.1.1.

The Flutter team describes this as "aggressive composability": All widgets should be composed of smaller widgets, all the way down to the smallest possible widget [66]. In fact, most standard Flutter widgets are based on the concept of aggressive composability. By creating small and focused widgets, the developer promotes code reuse. Having an arsenal of composable widgets allows for rapid future development, by creating large widgets through a composition of smaller ones.

8.2 State

In many cases, an application has to respond a certain way to user interactions. When flipping a switch within an application, the user expects the application to remember the action and update the user interface accordingly. Choices similar to this one can be described as state, a piece of information the application must somehow keep in memory and display to the user. State may change any number of times throughout the life-cycle of the application. Flutter addresses state through stateful and stateless widgets. A stateless widget is simply a widget with no state;

The contents of the widget will never change throughout the application lifecycle. Stateful widgets are able to remember state and rebuild themselves to reflect changes within the state.

8.3 Reactive Applications

An important programming paradigm to understand before reading about the implementation of this application is reactive programming. The paradigm revolves around programming systems through the use of asynchronous streams of data. As opposed to the imperative way of having to request data every time it is needed, reactive programming allows one to receive a continuous stream of data as it is added to the stream. By utilizing streams, one can ensure the user interface is always reflective of the underlying data. This concept is heavily utilized throughout the application. Nearly all data displayed throughout the application is provided by streams, allowing for dynamic and up-to-date user interfaces at a low cost.

8.4 Relevant Flutter Files

In simple use cases, Flutter developers can avoid touching any native Android or iOS code. For more complex projects, however, some modification is required. Throughout the development of the mobile application, the only need for native code were modifications of platform configuration files. In Android, this is done through a file named *AndroidManifest.xml*, whereas iOS uses a file called *Info.plist*. For the most part, changes revolved around adding permission requests. Apple is very particular about requesting permissions; If a permission is to be granted, it must be for a good reason. As such, every permission request has added transparent descriptions surrounding their usage.

Platform configuration files have also been modified to enable certain Flutter plugins. The plugin used for background location tracking, *background_locator* [67], requires several modifications to configuration files. For Android, one simply has to specify a service in *AndroidManifest.xml*. The iOS implementation requires the developer to enable *background modes* and *location updates* in XCode, followed by registering the plugin in the AppDelegate of the XCode project. *AppAuth* [68], the authentication plugin utilized by the application, requires the developer to specify URL schemes for authentication. This is specified in the Gradle build file for Android, and the aforementioned *Info.plist* file for iOS.

Project dependencies are defined in *pubspec.yaml*. This file is also responsible for defining project-specific settings, such as name, description, version and static assets. Furthermore, the project utilizes custom analysis and build options, which can be found in *analysis_options.yaml* and *build.yaml*, respectively.

8.5 Flutter Packages & Plugins

Flutter is written using Dart, a programming language developed by Google [44]. Like most programming languages, Dart allows for the use of packages and plugins to add new functionality to the language. In fact, essential functionality is not always implemented into Flutter directly. For instance, camera functionality is only provided as a standalone plugin. The Flutter team distinguishes between packages and plugins in the following way: "A plugin package is a special kind of package that makes platform functionality available to the app" [69]. By this definition, the aforementioned camera package would be considered a plugin, as it enables the application to utilize native camera functionality. This section will describe the most important packages and plugins the application makes use of, why they were chosen, and how they are used.

8.5.1 Provider

As an application increases in size, the process of managing state can increase in complexity. The "solution" to this problem has been to create packages for managing state. The Flutter development environment has provided a plethora of state management solutions, where each developer favors one over the other. The author chose to utilize *Provider* [70], a solution recommended by the Flutter team [71]. In comparison to other alternatives within the space, it requires less boilerplate code, and does not force the developer to develop in a specific way. Furthermore, *Provider* is quite simple. Opting to go for a more complex solution could potentially introduce additional complexity with no additional benefit.

Provider is a mix between state management, service location and dependency injection. It simplifies the process of utilizing *InheritedWidget*, a Flutter widget allowing a widget to expose information to descendant widgets. Utilizing such widgets are usually known as "lifting state up": Two widgets can utilize shared state stored in a parent widget. Usage of Provider within this application is further described in Sections 10.1 and 10.13.

8.5.2 Flutter_map

Reading and navigating a map is a vital piece of functionality in the application. In order for the application to function as intended, a solid map is essential. The Flutter team provides a package for Google Maps integration which cannot be used for this application, as it does not support custom tile providers nor offline map storage. The only map plugin supporting offline tile storage is *Flutter_map* [72], which offers more flexibility and customization than any other existing solution. In fact, it is the only available map plugin supporting the usage of tiles stored locally on the device file system.

8.5.3 Moor

The application must allow allow for users to perform trips without internet connectivity. In order to achieve such a task reliably, data must somehow be persisted to local storage. This application solves this problem by using a SQLite database, and is discussed further in Section 10.2. Flutter does not provide an official solution for working with databases, but third party packages are available. The author chose to utilize Moor [73], a database persistence library built on top of the most popular Flutter library for handling SQLite databases, *sqflite* [74].

Moor provides additional functionalities that makes working with SQLite databases easier. Defined database tables can be analyzed by Moor, which in turn generates Dart API's. This removes the need to write SQL queries by hand, expediting the development process. Furthermore, Moor automatically transforms query results into Dart types. It even allows for queries to be returned as streams, simplifying the process of synchronizing application and database state. In comparison to Moor, *sqflite* is verbose. It allows for much greater flexibility at the cost of development time, due to an abundance of boilerplate code. The author believes the flexibility trade-off to be worth it, as Moor appeared to cover any functionality the application might require.

8.5.4 RxDart

Streams are an important part of Flutter, and allows one to listen to a sequence of events as they appear. Flutter provides support for streams out of the box. However, as with many other modern programming languages, ReactiveX improves upon it. The application utilizes RxDart [75] to perform complex stream operations not covered by the standard streams library.

8.5.5 Dio

Being able to perform network requests is a crucial part of the application, and is something Flutter does not provide out of the box. Although Flutter provides the official *http* plugin, many prefer *Dio* [76] due to a larger feature set. This application chose *Dio* due to the support for request and response interceptors, automatic JSON parsing and simplified request timeouts.

8.5.6 Flutter_appauth

According to the Internet Engineering Task Force (IETF), mobile authentication is best performed through the use of external user-agents [65]. *AppAuth*, an open-source library for doing exactly this, is referenced within the recommendation. Thus, choosing the Flutter implementation of *AppAuth* could be considered a sound choice. *Flutter_appauth* [68] simply provides a wrapper around the iOS and Android implementations of *AppAuth*, and allows the authentication process to be

performed in an external user agent. It handles the entirety of the authentication process in the application. The authentication implementation for the application is described in Section 10.8.

8.5.7 Flutter_secure_storage

Most of the data the application stores is not sensitive. However, some data should be kept secret from the user. More specifically, access tokens and refresh tokens should not be known to any other entity than the system. The application enables this by utilizing *Flutter_secure_storage* [77], a plugin exposing API's to the native implementations of secure storage for the underlying operating system.

8.5.8 Camerawesome

The official Flutter camera plugin was not production-ready during development. Due to several crucial bugs and missing features, the author opted to choose a third-party plugin, *Camerawesome* [78], for accessing the camera. At the time, it appeared to be the most flexible and well maintained option, and is simply used for accessing native camera functionality.

8.5.9 Geolocator

The user location is accessed through the use of *GeoLocator* [79], a plugin providing easy access to the underlying native location implementations. It allows developers to listen to a stream of location updates, and many other utility functionalities that expedites the development process. Although other options are present, none are as well maintained and documented as *GeoLocator*. Location usage for the application is described in Section 10.10. Furthermore, problems surrounding location is discussed in Section 12.2.

8.5.10 Background_locator

Location tracking when the application is inactive or killed is handled through the use of *background_locator* [67]. It allows for one to spawn an isolated process with the sole responsibility of receiving location updates. The mobile application will encapsulate library usage into a service capable of stopping and starting background location tracking whenever the application can no longer do so itself.

Chapter 9

Application Architecture

Having a well-planned architecture is key when it comes to designing modifiable and extensible applications. This section describes the different patterns utilized by the mobile application to allow for a modular and scalable solution. Finally, it presents a set of diagrams describing the overall structure of the architecture.

9.1 Architectural Patterns

Bass et al. defines architectural patterns and tactics as "... ways of capturing proven good design structures, so that they can be reused" [28]. By relying upon tried and true knowledge, one is able to create better end products. This section will describe the architectural patterns used throughout the mobile application.

9.1.1 Onion Architecture

The Onion architecture was defined by Jeffrey Palermo as a direct response to the tight coupling layered architectures often led to [80]. In the layered architecture, each layer depends on the layer beneath it, in addition to a set of common infrastructural and utility functions. The Onion architecture combats this by proposing a different way of coupling layers together. Instead of the top-down approach the layered pattern utilizes, the onion architecture specifies layers as rings, where a layer is only allowed to access resources within itself or layers closer to the core of the "onion". Layers in the Onion architecture can be separated in two: the application core and the outer layer. The original representation of the architecture is shown in Figure 9.1.

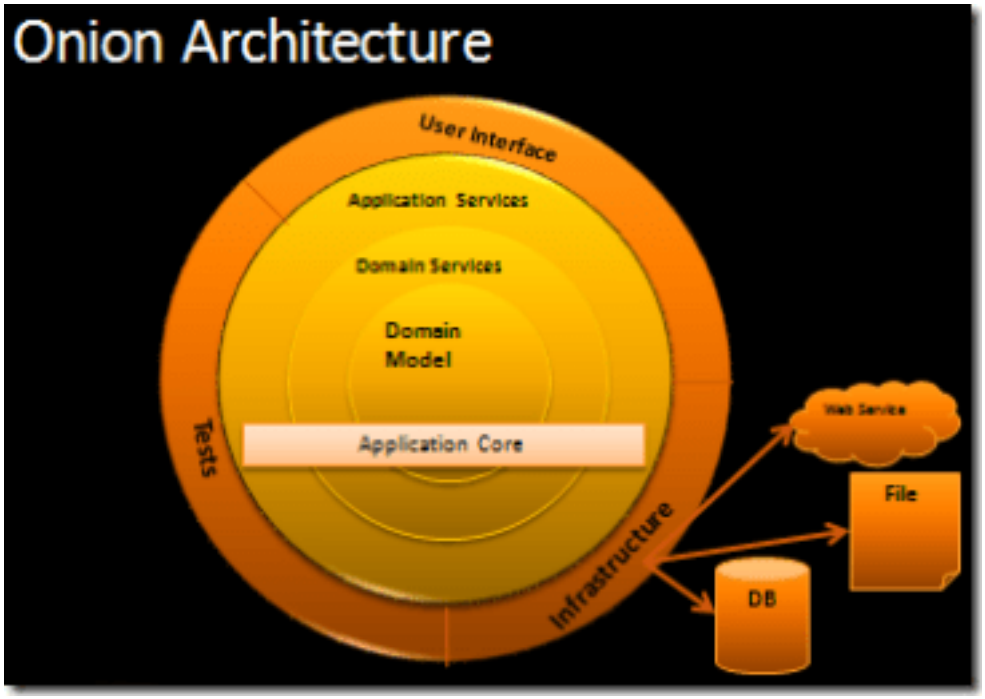

Figure 9.1: The Onion Architecture, as presented by Palermo [80].

The application core consists of several layers, and defines what entities, business logic and application logic the application requires in order to function. The innermost layer, the *domain* layer, defines domain entities, business logic and interfaces for functionality a domain relates to or depends on. Such an interface could for example be persistence related. The inner layer only depends on itself, and rarely changes. Following the domain layer is the *application* layer, a layer concerned with application rules, a set of actions one must perform in order to successfully perform an application use case. Such rules are usually implemented by orchestrating domain services in order to achieve the specified end result. The application core specification is not rigid, and can contain more layers if required. However, for the purposes of this application, the aforementioned ones proved to be sufficient.

The outer layer is home to rapidly changing components. Figure 9.1 provides user interfaces, tests and infrastructure components as examples, something that is reflected in the actual implementation of the architecture. Anything stored in this layer is allowed to access all layers within the application core. The *user interface* aspect of the example contains things like views, view models, widgets and interaction logic, whereas the *test* example simply contains tests for the entire application.

Infrastructure, on the other hand, is a much more abstract concept. It contains a wide range of responsibilities, but no business-critical logic. Any business or application logic dealing with I/O will be defined here. For instance, if an entity in the *domain* layer requires persistence, an interface for the persistence requirements would be defined within the *domain services* layer. The actual persistence implementation would be defined in the infrastructure layer. In this case, the infrastructure is coupled to the domain, not the other way around.

The Onion architecture allows for loosely coupled layers. By keeping rapidly changing components in the outermost layer and requiring all dependencies to point towards layers closer to the core, change propagation is kept to a minimum. Changes within the application core will propagate outwards, but such occurrences are more rare. The separation of concerns between layers also allows one to more easily switch out any external dependencies or frameworks, as it should not affect the application core. Furthermore, the usage of the Inversion of Control principle allows for low coupling between dependencies. By depending on interfaces instead of actual implementations and avoiding any instantiation logic within dependents, the ease of which one can swap components or mock them for testing is greatly increased. The author believes the Onion architecture to bring enough flexibility to future-proof the application, both in terms of extensibility and scalability.

9.1.2 Model-View-ViewModel

The separation between user-interface code and business logic is handled with the Model-View-ViewModel (MVVM) pattern. Whereas traditional Model-View-Controller and Model-View-Presenter approaches create a tight coupling between views, models and controllers, MVVM focuses on separating the view and the business logic entirely [81]. The concept is based on three components: Models, Views and View Models.

The view is responsible for defining user interfaces and handling user interactions, and contains as little logic as possible. Every view has a corresponding view model, which can be viewed as a handler for the state of a view. View Models provides a set of properties the view can bind to. Whenever the state of a view model changes, it will emit an event the view can listen to. This allows for views to dynamically rebuild themselves whenever the state of the view model changes. View models should contain as little logic as possible; An ideal view model should primarily be concerned with calling models and maintaining the state of a view.

View Models are also responsible for interacting with the models throughout an application. In simple terms, a model encapsulates the data of the application [81]. Views are able to interact with the models through the use of functions exposed by the view model. Models have no knowledge of view models, and view models have no knowledge of views. Both Models and View Models can be considered to be observable objects simply emitting events without knowing who will receive them.

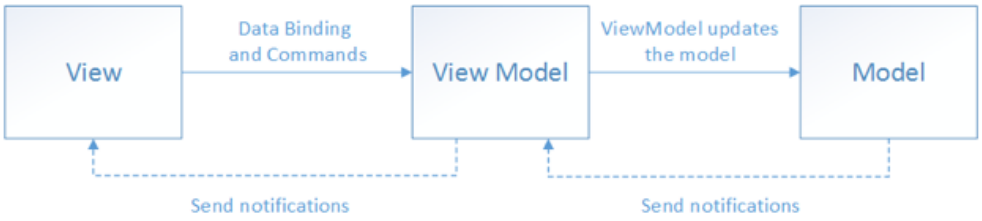

Figure 9.2: The MVVM pattern, as described by Microsoft [81].

The relations between the components of the MVVM pattern is shown in Figure 9.2. This description of the pattern mostly aligns with the implementation of the pattern in the application. The only change between the figure and the implementation is that view models does not update models directly. Such logic is instead handled by intermediaries located in the *application* and *infrastructure* layers of the *Onion Architecture* pattern.

9.2 Design Patterns

Many of the problems one faces when developing software has predefined solutions to them, as they have been encountered many times before. Such solutions are oftentimes called design patterns, a term coined by Gamma et al [82]. This section will describe the design patterns used throughout the application, and reason as to why they were chosen.

9.2.1 Service Locator

The service locator pattern describes an object with the knowledge of how to retrieve instances of system services [83]. This is usually implemented through the use of an IoC container, in which one can instantiate and store instances of an object. The container is able to receive requests for the stored instances and return a reference to it. Whenever a service is needed, the application can simply call the service locator to get a reference to the needed interface, without necessarily knowing what the underlying implementation is. In the case of this application, the pattern is used to provide services and repositories to view models without instantiating them inside the view model.

9.2.2 Dependency Injection

Dependency Injection is another pattern used for achieving Inversion of Control (IoC). The main difference between Dependency Injection and the Service Locator pattern is one of knowledge. Classes using the Service Locator pattern are fully aware of the existence of the IoC container. When using dependency injection, on the other hand, the knowledge of the IoC container is completely removed [83].

Within the context of this application, dependency injection is preferred wherever possible, as it simplifies testing. Testing classes relying on a service locator will require one to create a service locator and fill it with mock versions of the required dependencies. With dependency injection, on the other hand, mocked instances of the required dependencies can instead be instantiated directly. Service location is only used in the application views; Throughout the rest of the application, dependencies are provided through injection.

9.2.3 Data Access Object

A Data Access Object (DAO) provides an abstraction layer on top of a persistence mechanism [84]. Instead of providing direct access to a persistence solution, the DAO pattern explicitly states that all persistence interactions should be performed through a list of API calls the DAO exposes through an interface. Using Data Access Objects allows the developer to hide implementation details of any' particular persistence solution from the rest of the system. If one were to switch the persistence implementation, the DAO would ensure that no modifications would have to be made for the rest of the system. All entity persistence within the application is performed through the use of Data Access Objects.

9.2.4 Repository

Hieatt and Mee defines the Repository pattern as such: "A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection" [85]. Whereas a DAO acts as a persistence abstraction of a specific entity, a repository acts as an abstraction of the storage of an aggregate root: A collection of domain entities that can be treated as a single entity. A repository implementation can, and often does, use several Data Access Objects to achieve its purpose. Each aggregate root within the final application has its own repository.

9.2.5 Observer

In order for the view to listen for changes in the view model state, a solution for notifying listeners must be created. Such solutions are oftentimes implemented with the observer pattern, in which an observable class is allowed to have zero to many listeners who will receive notifications of any events the observable object emits [82]. In the application, every view model is by default an observer, whereas each view is a listener. This allows views to rebuild themselves whenever a change in the view model state occurs. Furthermore, the streams used throughout the application are largely based on the observer pattern.

9.2.6 Value Object

Comparing the equality of two objects can be achieved in several ways. In many languages, objects are equal when they reference the same object. Another way to look at object equality is to compare the actual values that comprises the object. In this context, the object can be described as a value object: An immutable object that should be equal to any other objects with equal values [86]. The application will utilize value objects wherever equality of value is more correct than referential equality.

9.2.7 Singleton

According to Gamma et al, the Singleton pattern is used to "Ensure a class only has one instance, and provide a global point of access to it" [82]. The pattern prohibits the instantiation of new class instances; Instead, the class is only instantiated when first accessed. If one were to create a new reference to the class later on, the previously created instance would be returned. Since its inception, the Singleton pattern has been criticized for being an anti-pattern. Safyan claims that Singletons reduces the testability and modifiability of systems [87]; Similar sentiments are shared across the web. The application only utilizes the pattern indirectly. Third-party packages oftentimes exposes Singleton instances of their provided functionality. When used in the application, Singleton usage is hidden in implementation details behind interfaces, allowing for one to easily switch to a different alternative if necessary.

9.2.8 Facade

Oftentimes, the consumer of a subsystem is not interested in complex implementation details. If given the choice, it is reasonable to assume a simple set of API's would be the preferred choice over having to grasp how a system works. The facade pattern describes the process of restricting access to complex subsystems through the use of a simpler, shared interface [82]. Many of the subsystems throughout the system are not accessed directly, but instead through the use of interfaces. Repositories are a good example; A repository interface provides no details surrounding how an entity is persisted or mutated.

9.2.9 Mediator

The mediator pattern allows one to orchestrate a set of object functionalities in such a way that a specific goal is achieved [82]. Having direct dependencies between objects would create a tight coupling; By making the objects completely independent of one another, and instead opting to create a separate object orchestrating the relationship between them, the system will have a lower coupling. Complex use cases requiring interacting with different application responsibilities will utilize the mediator pattern to orchestrate the interactions.

9.3 Architectural Description

Project structure oftentimes correlates with the architectural patterns the system employs. Since the mobile application utilizes the onion architecture, separating source code by layer would be sensible. Code is mainly divided into four folders: *domain*, *infrastructure*, *application* and *presentation*. Each layer is subdivided by domain; A *farm* folder is present in all layers, albeit with different concerns. This approach allows clear separation between layers, while still ensuring maintainable folder sizes.

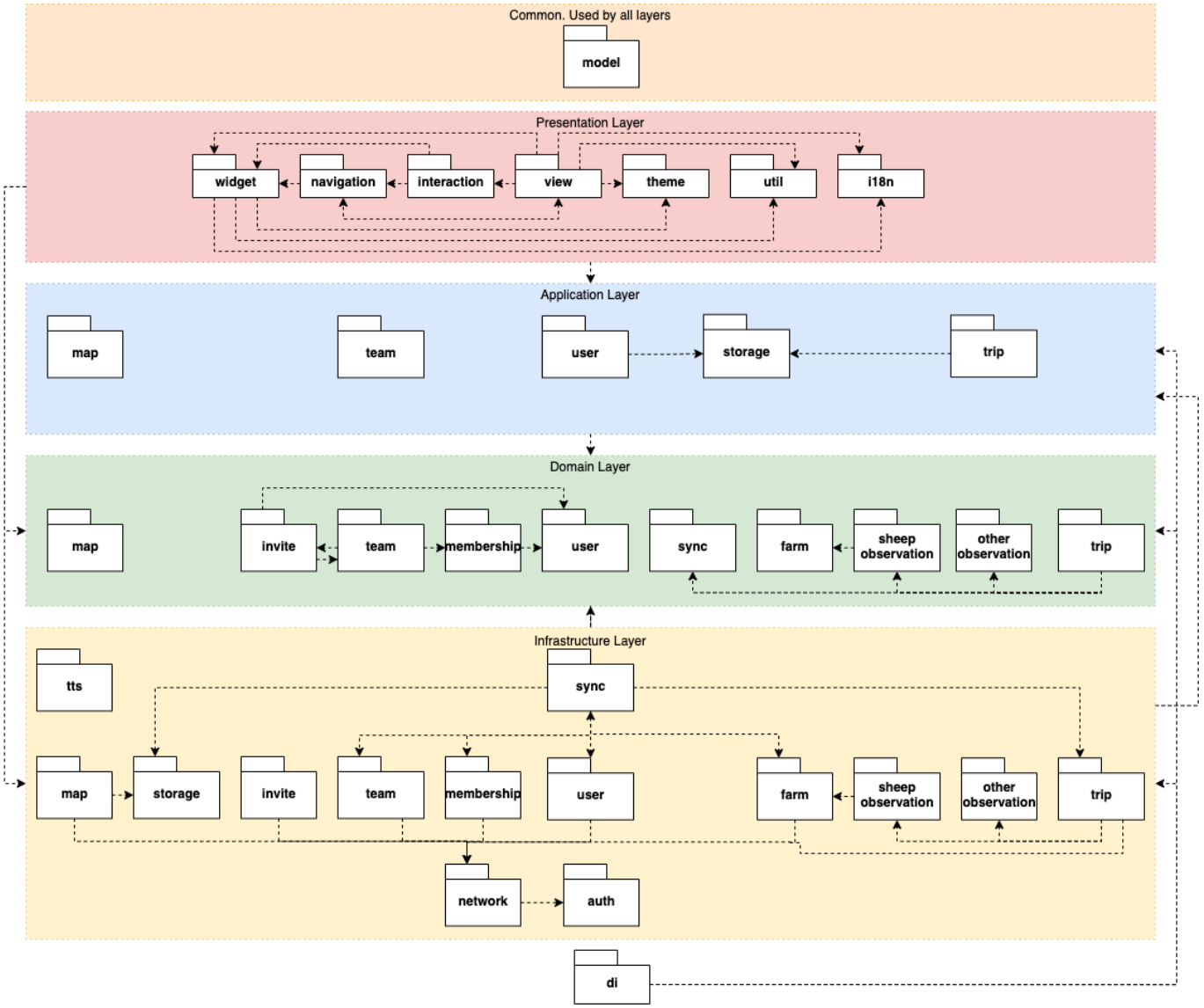

Figure 9.3: Package diagram of application structure.

A package diagram of the project structure is shown in Figure 9.3. The figure only displays relations within layers to avoid clutter. Cross-layer dependencies are abstracted away into the arrows shown in the center and sides of every layer.

At the very core of the architecture lies the *domain* layer. It stores domain entities, business logic and interfaces defining domain-specific service requirements. A folder within the *domain* layer can contain any of the following folders:

- **entity** Definitions of persistable objects.
- **enum** Special classes with a set of predefined values.
- **interface** Interfaces for repositories and business logic the *infrastructure* layer should implement.
- **vo** Value Objects: A simple, immutable object compared to other value objects by the values stored within, as opposed to by reference.
- **model** Objects only utilized throughout the domain.

Following *domain* is the *application* layer, which mostly deals with receiving data from the *presentation* layer, and orchestrating resources in order to achieve the desired end result. The layer only contains service interfaces and implementations.

The *infrastructure* layer is part of the outermost layer of the onion. Any code relating to external services, storage or platform operations are stored here. A folder in the *infrastructure* layer can contain any of the following sub-folders:

- **dao** Data Access Objects for fetching locally and remotely stored data.
- **model** Database table definitions.
- **repository** Implementations of repository interfaces defined in *domain*.
- **service** Service implementations relating to external resources.

Finally, the *presentation* layer contains code relating to views, view models, widgets, interaction services, navigation services and themes. The presentation layer is not divided in sub-folders by domain, but rather by concept:

- **i18n** Code related to the internationalization of the user interface.
- **interaction** Services used to display snack bars, alerts and dialogs.
- **navigation** View names, navigation services and navigation arguments.
- **theme** Theme definitions for light and dark mode.
- **util** Various utility functions used in several views.
- **view** Views, view models and domain-specific widgets grouped by domain.
- **widget** Generic widgets used throughout the application.

Source code can also be found in two additional folders. The *di* folder is responsible for instantiating injectable resources, whereas the *common* folder contains models not belonging to any particular domain.

Figure 9.4 displays a generic vertical slice of the architecture, and describes the relationships between classes. The presentation layer contains domain-specific widgets, a view and a corresponding view model. View models interact with the underlying layers through the use of an entity service providing functionality for domain-specific use cases. This is usually achieved through the use of the entity repository, which is used to query locally or remotely stored entities. Whether an action must fetch or mutate entities locally or remotely is decided within the repository.

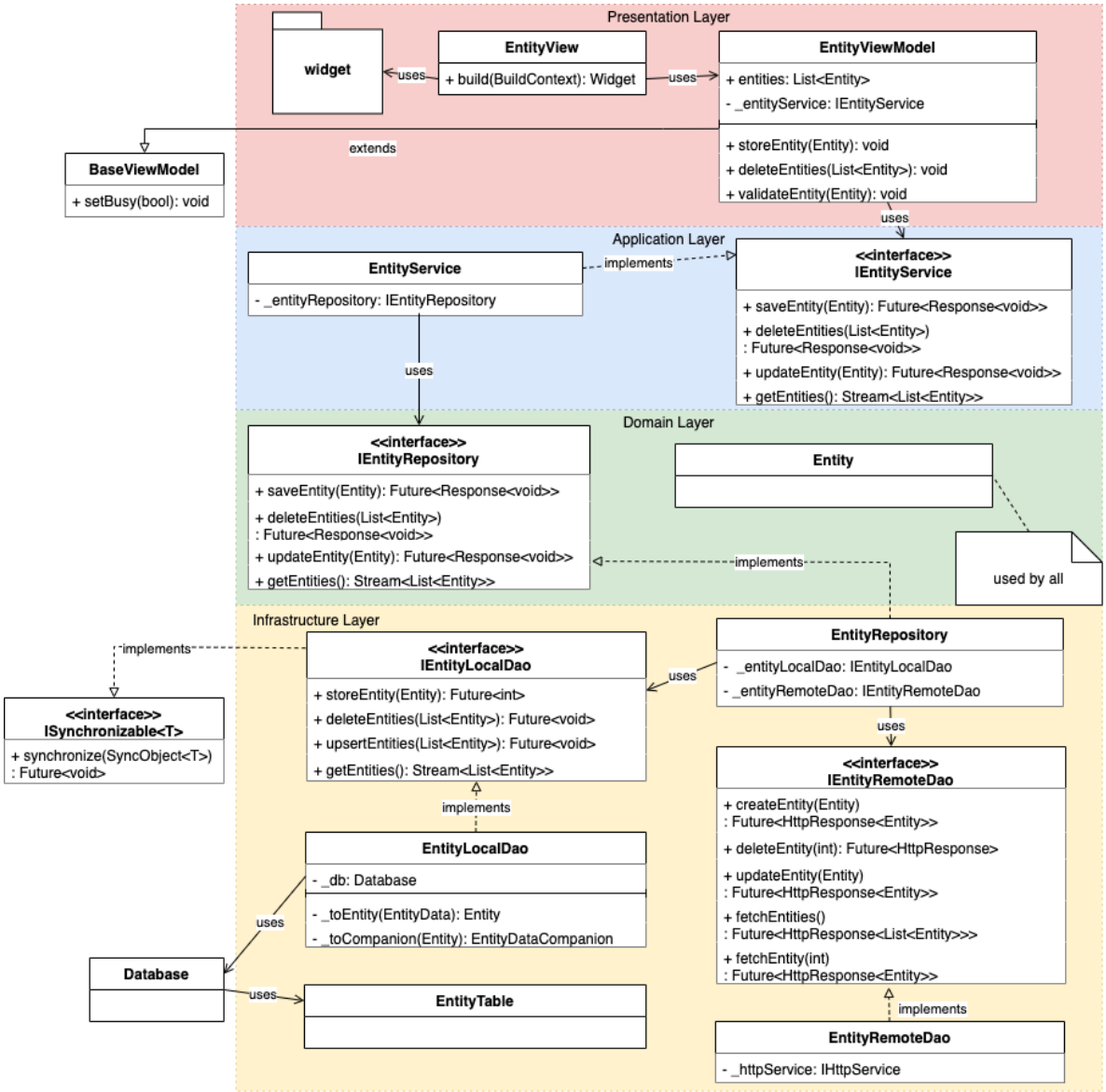

Figure 9.4: Class diagram of a vertical slice of the application architecture.

Repository implementations usually references DAO's for both local and remote data access. In Figure 9.4, these are represented by *EntityLocalDao* and *EntityRemoteDao*, respectively. Local DAO's utilize the *Database* for entity access, whereas remote DAO's utilizes a service for performing HTTP requests. In the case of this diagram, the local DAO also implements the *ISynchronizable* interface. This indicates that the entities stored locally on the device should be synchronized with the server. The database uses the *EntityTable* to generate Dart API's for use within the local DAO.

Repositories, services and DAO's all have interfaces and corresponding implementations. The actual implementations are not referenced throughout the entire application; They are only used to instantiate instances of an interface for injection later-on.

Some simplifications of the vertical slice of the architecture have been made to ensure the readability of the diagram. Views and view models oftentimes utilize several resources located within the presentation layer; Navigation services, interaction services, generic widgets and custom themes are all common things views and view models utilize. Furthermore, the usage of the actual entity is not mapped, due to the fact that every class expect for the *EntityTable* uses it to some extent.

Both sheep and other observations are comprised of several sub-domains. A decomposition of the sheep observations folder for the *infrastructure* and *domain* layers are shown in Figure 9.5. Relations between layers have been removed once again in order to avoid clutter. In short, a DAO will always utilize its corresponding entity. The figure shows that every entity used within the sheep observation entity has a corresponding DAO interface. Directly accessing DAO's through these interfaces are not encouraged. Instead, one is encouraged to store sheep observations through the sheep observation repository, which in turn utilizes the different DAO's. The actual implementation of the DAO's is abstracted away in the diagram. Instead, the *EntityLocalDao* is shown. In order to keep the diagram at a manageable size, this abstraction is meant to represent implementations of every DAO interface. The same applies to *EntityTable*, which is meant to represent the database table definitions for each entity.

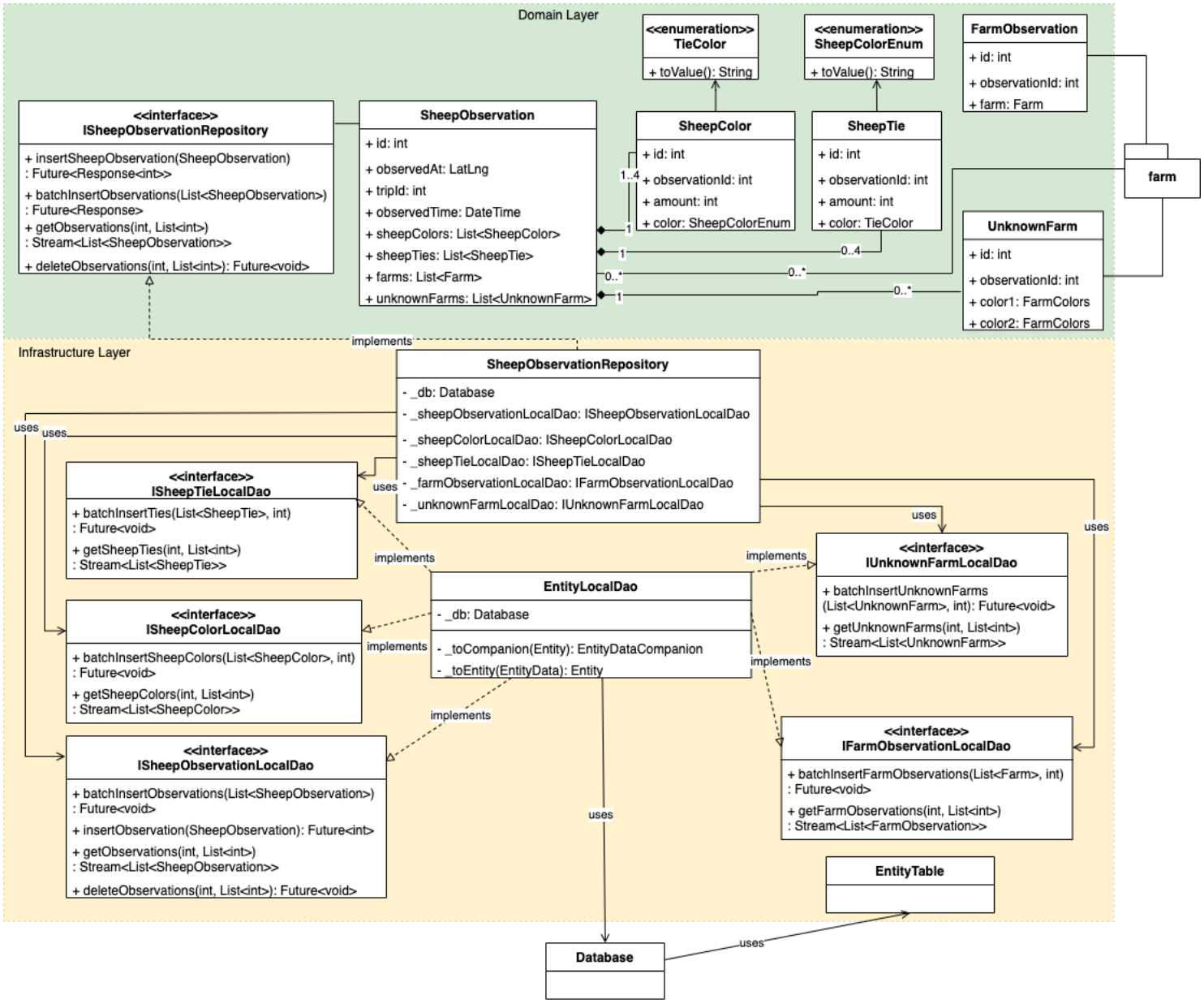

Figure 9.5: Class diagram of *domain* and *infrastructure* layers of the sheep observation domain.

Another important concept to decompose is *views*. As previously stated, views are divided in terms of what domain they belong to. Thus, a folder within a specific domain might contain several view folders. A good example of this is the *team* folder, which contains a total of eight views. Figure 9.6 details all of the views available throughout the application, and how one can navigate between them. The figure is slightly simplified. It does not display the flow for navigating back to the previous view; However, one can always navigate back to the previous view, with the exception of the authentication view. The only way to navigate back to the authentication view is by logging out.

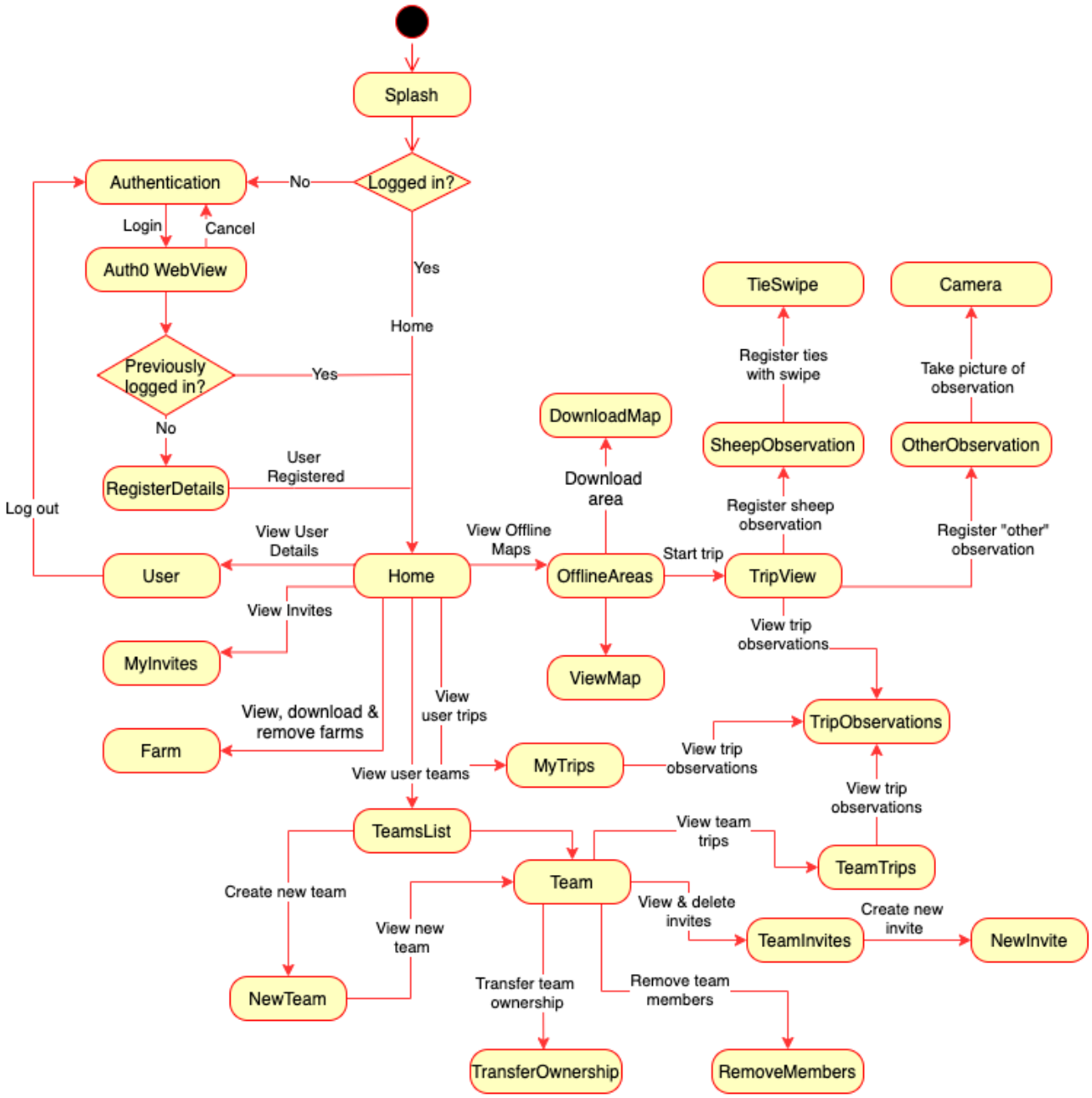

Figure 9.6: Navigating between different views.

Chapter 10

Application Implementation

This chapter describes the most important implementation details of the mobile application. Each section covers a vital part of the application through textual descriptions and code snippets.

10.1 MVVM implementation

The MVVM pattern is implemented through the use of the *Provider* package and *ChangeNotifier*, the Flutter equivalent of an observable object. *ChangeNotifier* exposes a set of functions for adding and removing listeners, and notifying them of any changes. Every view model extends the *ChangeNotifier* class.

Whenever a user navigates to a new screen, Flutter adds a new view to the stack of the Flutter Navigator. Thus, the view has to instantiate its corresponding view model. The base of any view is a *ChangeNotifierProvider*, which the *Provider* documentation describes as an object that listens to a *ChangeNotifier*, exposes it to any of its descendants and rebuilds whenever the *ChangeNotifier* notifies its listeners of a change [70].

ChangeNotifierProvider expects two parameters: a *ChangeNotifier* and a child widget. The child widget will be recreated whenever the *ChangeNotifier* changes. By passing a view model as the *ChangeNotifier* to provide, the child widget will be able to access the view model and rebuild whenever it changes.

The aforementioned usage of *ChangeNotifier* and *Consumer* is encapsulated in a widget named *BaseWidget*. Every widget utilizing a view model has this widget as their root widget. *BaseWidget* exposes four variables: *model*, *onModelReady*, *builder* and *child*. An excerpt of the parameter definitions of *BaseWidget* is shown in Listing 10.1.1, whereas the build function is shown in Listing 10.1.2.

Code listing 10.1.1: Parameters and type definition of BaseWidget.

```
class BaseWidget<T extends ChangeNotifier> extends StatefulWidget {
  final Widget Function(BuildContext context, T model, Widget child) builder;
  final T model;
  final Widget child;
  final Function(T) onModelReady;

  ...
}
```

The model variable expects an instantiated *ChangeNotifier* to be used as a view model for the widget. Instantiation logic can be passed into the *onModelReady* variable. *onModelReady* exposes the instance passed into the *model* variable, allowing one to call its methods. If provided, the *onModelReady* function will only be called when the view model is instantiated.

The *builder* variable expects a function returning the widget *BaseWidget* should display, and is called when the corresponding view model emits a change. Within the provided *builder* method, one is able to access all public properties of the corresponding view model. Furthermore, it exposes the *child* variable, a widget which will not change during the lifespan of the view. This is mostly used for optimization purposes, as the *child* will not have to be rebuilt every time the *builder* widget is.

Code listing 10.1.2: How the BaseWidget is built.

```
Widget build(BuildContext context) {
  return ChangeNotifierProvider<T>(
    create: (context) => model,
    child: Consumer<T>(
      builder: widget.builder,
      child: widget.child,
    ),
  );
}
```

Even though the *BaseWidget* provides a nice simplification of view and view model instantiation, the author believed a further simplification to be beneficial. As such, the *BaseView* widget was created. *BaseView* allows the developer to specify commonly used widgets within a view through the use of parameters, all whom have default values. This allows for a more uniform look for each view. Furthermore, it hides several layers of code indentation caused by wrapper widgets, increasing the readability of the source code. The definition of the *BaseView* class can be seen in Listing 10.1.3, and a rudimentary view and view model is shown in Listing 10.1.4.

Code listing 10.1.3: BaseView definition.

```
class BaseView<T extends BaseViewModel> extends StatelessWidget {
  final T model;
  final Widget Function(BuildContext, T model, Widget) builder;
  final Widget child;
  final Function(T) onModelReady;
  final bool safeAreaTop;
  final bool safeAreaBottom;
  final bool safeAreaLeft;
  final bool safeAreaRight;
  final EdgeInsets basePadding;
...
}
```

Code listing 10.1.4: A rudimentary view and view model implementation.

```
class TestViewModel extends BaseViewModel {
  String get exampleText => "hello";
}

class TestView extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return BaseView<TestViewModel>(
      model: TestViewModel(),
      builder: (context, model, child) => Text(model.exampleText),
      onModelReady: (model) => print(model.exampleText),
      basePadding: const EdgeInsets.all(20),
      safeAreaTop: false,
      safeAreaBottom: false,
    );
  }
}
```

10.2 Application Database

The application utilizes a SQLite database to locally store data the application requires in order to function without internet access. This section describes why the project uses a relational database, and why it uses *Moor* over *Sqflite*. Finally, it will describe how the database is structured and accessed.

In terms of database types, one can choose two very different approaches for mobile: A relational SQL database, or NoSQL. The traditional approach to implementing databases is the relational approach, allowing the developer to store data in predefined schemas that can have relations to one another. It allows for a great amount of query variability, and has excellent, battle-tested tooling. Most relational databases uses the Structured Query Language (SQL) language for querying data.

NoSQL, on the other hand, is a more modern approach, and not as easy to describe. In essence, NoSQL (Not only SQL / Not SQL) is any database approach that stores data in another format than relational tables. The way data is read and modified differs drastically from each implementation and type of NoSQL database.

Both approaches are used heavily within the Flutter community. For NoSQL implementations, many choose to use Firebase, Google's approach to simplifying application development. It provides a plethora of services interlinked with other Google Cloud Platform (GCP) services. In terms of storage, Firebase provides Realtime Database [88] and Cloud Firestore [89], whom are both NoSQL databases supported by Flutter. Furthermore, the solutions provides offline storage and synchronization out of the box.

Several packages are available within the relational space as well. However, only two have gained major traction: Sqflite [74] and Moor [73]. Sqflite provides the necessary classes and functions to interact with a SQLite database. The developer is expected to define all queries and transformations themselves, allowing the developer a great deal of flexibility with regards to implementation. On the other hand, it introduces a lot of boilerplate code, increasing the overall development time.

Moor, on the other hand, is a database library built on top of Sqflite. It provides additional functionality that removes a lot of the boilerplate associated with using Sqflite. This is primarily solved through the use of code generation. Moor allows the developer to generate Dart API's through database table definitions. The generated code provides type-safe transformations and queries. Furthermore, Moor allows users to utilize streams, a central concept in Dart. This would allow for an easy way for the user interface to always show the most recent data from the database in the user interface.

Choosing any Firebase approach could be beneficial, as it would remove the need for integrating synchronization support for both the mobile application and the server backend. The possible trade-off is that the stored data would not be as structured as a relational database. Furthermore, choosing a NoSQL solutions cannot provide as complex and granular querying capabilities as that of relational databases. The querying potential of relational databases, and the ability to utilize it to use the data in different ways, proved to be the deciding factor in the choice of database technology. As such, the author chose to use a relational SQL database.

After having experienced Moor and Sqflite, the author preferred Moor. The loss of flexibility the automatically generated code introduced never became a problem, as Moor proved to contain functionality for every requirement. Furthermore, it drastically reduced development time, resulting in nearly no boilerplate code. The maintainer of the project proved to be very active, and continuously updated the framework.

10.2.1 Moor Usage

Moor allows for the creation of database tables through both SQL and code. The author chose to use code, as it would allow one to separate each table into their own file. An example of this approach is shown in Listing 10.2.1.1, where the Farm table is defined through Dart code.

Code listing 10.2.1.1: Defining the farm table in Dart code.

```
class Farm extends Table {
  IntColumn get id => integer()();
  TextColumn get color1 => text()();
  TextColumn get color2 => text()();
  TextColumn get name => text().customConstraint("collate nocase")();
  DateTimeColumn get createdAt => dateTime()();
  DateTimeColumn get modifiedAt => dateTime()();

  @override
  Set<Column> get primaryKey => {id};
}
```

The actual database object is defined as a class, and is responsible for the creation of the database, the database connection and any other operations directly associated with the actual database. The object instance is made available to the application through the use of *Provider*, allowing it to be injected into repositories and services when needed.

Moor will only generate code for the tables specified in the Database class. Generation is done through the Flutter *build_runner*, a Dart package responsible for automatically generating Dart code [90]. The developer can start the *build_runner* with the command *flutter pub run build_runner*. One can also append the *watch* and *delete-conflicting-outputs* arguments, allowing for code regeneration every time the project is saved.

10.2.2 Table Structure

An entity relationship diagram of the table structure is shown in Figure 10.1. The following paragraphs provides short descriptions of each table. All fields are non-nullable, unless explicitly stated otherwise.

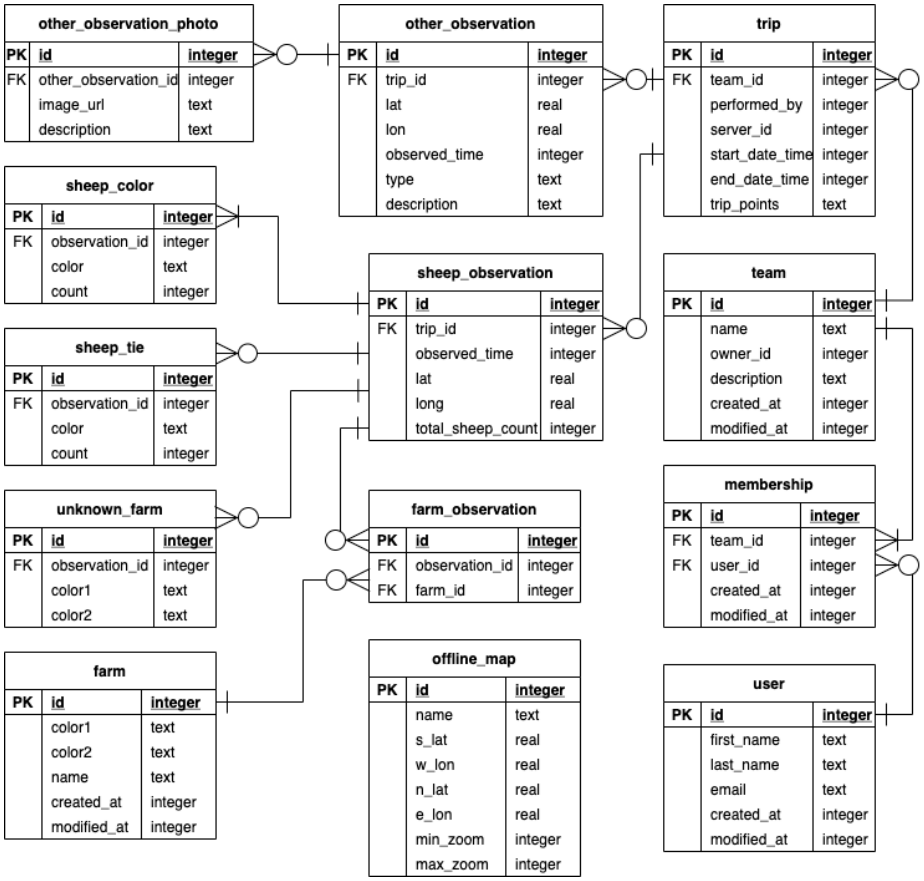

Figure 10.1: Entity relationship diagram of Application database.

offline_map Contains maps downloaded to the device. It specifies the map name, the coordinate bounds of the downloaded map and what zoom levels have been downloaded. The map name has a "NOCASE" collation, allowing queries to ignore letter casings when filtering rows by name.

sheep_color Contains the number of sheep with a specific wool color within a sheep observation, and references the sheep observation it belongs to. It is automatically deleted when the referenced sheep observation is deleted.

sheep_tie Contains the number of sheep with a specific tie color within a sheep observation, and also references the sheep observation it belongs to. It is automatically deleted when the sheep observation is deleted.

farm_observation A bridge table used to associate a known farm to a sheep observation. It references the sheep observation it belongs to and the observed farm, and is deleted if either of them are deleted.

unknown_farm Used for storing farms that have been observed in a sheep observation, but is not available as a known farm. It references the sheep observation it belongs to, and is deleted alongside it.

farm Specifies the name and identifying colors of a farm, and is used for storing known farms. Farm names have a collation of "NOCASE". Any table row can be referenced in zero to many *farm_observation* rows.

other_observation Contains all observations not concerned with live sheep. The table specifies the type of observation, where and when it was observed, and a description of the observation. It references the trip it belongs to, and is deleted when the trip is. Can be referenced in zero to many *other_observation_photo* rows.

other_observation_photo Contains an image reference and an optional image description. It references the observation it belongs to, and is deleted alongside it.

sheep_observation Describes a sheep observation within a supervision trip. A row specifies when and where the observation occurred and the total number of sheep within the observation. It references the trip it belongs to, and is deleted alongside it. It is referenced at least once in *sheep_color* rows, and zero to many times in *sheep_tie*, *sheep_color*, *observation_unknown_farm* and *observation_farm*.

trip Describes what team the trip belongs to, whom it is performed by and when it started and ended. *server_id* is nullable, and is set when the trip has been sent to the server. *end_date_time* is also nullable, and is set when the trip ends. *trip_points* is a serialized list of latitude and longitude pairs, and is set when the trip ends. The table references the team the trip belongs to, and is deleted alongside it.

team Specifies the name, owner ID and description of a team. It is referenced in at least one *membership* row. The team name has a "NOCASE" collation for ignoring letter casing during queries.

membership Describes what team a user is a part of. The table references both a team and a user, and is deleted whenever one or the other is deleted.

user Details the first and last name of a user, along with their email address.

10.3 Data Access Objects

Data Access Objects come in two different flavors: remote and local. Remote DAO's are responsible for mutating the entities stored server-side, whereas the local ones are responsible for mutating the local database. Both remote and local

DAO's are only accessed through the interfaces they implement, leaving the actual implementation details completely encapsulated within the DAO. All knowledge of DAO's are encapsulated within the infrastructure layer; No other layers are dependent on, or aware, of the existence of any DAO.

10.3.1 Local Data Access Objects

Every database table has a corresponding local DAO. The DAO's are defined as classes whom implement an interface specifying what operations the DAO must support. The actual implementation is highly dependent on the code generated by Moor, which allows for type-safe queries that can be developed and altered quickly.

Database tables only used within aggregate roots barely requires functionality within their respective DAO's. Currently, this applies to entities used within observations. These DAO's only expose functionality for getting and batch inserting entities. "Select" functions allows the developer to specify a set of parameters to constrain the result. Furthermore, query results are returned as a stream, meaning any modifications to the database will result in a new set of values being emitted.

Batch insertions expect to receive a list of entities that should be inserted within the same transaction. If one of the insertions within the transaction fails, all insertions are reverted, ensuring the database remains in a valid state. If such an error occurs, it will bubble up to the aggregate root that initiated the insertion, leading to the entire insertion being rolled back.

More advanced DAO's usually exposes functionality for entity deletion, with support for entity specific parameters. Furthermore, many of the advanced DAO's favors the usage of upserts instead of inserts. An upsert will, much like an insert, insert a new entity into a table. The major difference between the two is that an upsert updates the entity if it already exists, where an insert would simply throw an error. An example of batch upserting is shown in Listing 10.3.1.1, in which a list of farms are transformed into insertable Moor-objects and persisted to the database.

Code listing 10.3.1.1: Upserting a list of farms.

```
Future<void> upsertFarms(List<Farm> farms) async {
    return await batch(
      (batch) => batch.insertAllOnConflictUpdate(
        db.farmTable,
        farms.map(_toCompanion).toList(),
      ),
    );
}
```

Upserting simplifies the synchronization process, as one no longer has to check whether an object already exists before deciding if it should be inserted or updated. Synchronization is discussed further in Section 10.5. Upserts are only available for synchronizable entities. If an object is not synchronized, it provides an insert

instead. Trips are the exception to this rule. Most entities throughout the system receive an unique identifier when being persisted to the server, ensuring the uniqueness of the identifier throughout the entire system. This approach works well with network-reliant entities. Trips, on the other hand, are first created on the device. Although the uniqueness of the trip identifier is guaranteed within the context of the application, no such promises can be made for the system as a whole. In order to avoid identifier collisions, locally stored trips are instead updated to contain the identifier they receive on the server-side as well as their original identifier.

10.3.2 Remote Data Access Objects

Remote DAO's provide the application with the ability to mutate the state of the server database. These DAO's are implemented using the YAGNI (You ain't gonna need it!) principle: Functionality should only be implemented when it is actually required [91]. Thus, some entities do not have their own remote DAO. The ones that do require it only implements required functionality.

As previously stated, remote DAO's exposes a subset of Create, Read, Update, Delete (CRUD) functionality. Every implementation depends on an HTTP service, which handles the entirety of the request; The DAO implementation only prepares the request data and parses the response, as shown in Listing 10.3.2.1. HTTP is further discussed in Section 10.14.

Code listing 10.3.2.1: GET request for remotely stored invite.

```
Future<HttpResponse<Invite>> getInvite(int id) async {
    var result = await _httpService.getRequest("${APIEndpoints.invites}/$id");
    if (!result.success) {
      return result.convert<Invite>();
    }
    try {
      return HttpResponse<Invite>(
        result.statusCode,
        data: Invite.fromJson(result.data),
      );
    } catch (e) {
      return HttpResponse<Invite>(
        HttpStatus.unprocessableEntity,
        error: "Failed to parse invite: $e",
      );
    }
}
```

10.4 Repositories

Repository implementations are stored within the *infrastructure* layer. Their interfaces, on the other hand, are stored within the *domain* layer. This allows for

the *domain* layer to be completely independent of any other layer, and hides the knowledge of where and how entities are actually stored.

The application repositories orchestrates entity DAO's to react to user interactions throughout the application. It abstracts away the knowledge of whether a user action results in a changes within the local database, the remote database or both databases. Furthermore, it is responsible for collecting all entities used within an aggregate root. It does this by utilizing several DAO's to collect streams for each entity within the root. Finally, it combines the streams into a single stream of the root entity, which will emit a new value if any of the underlying entity streams emits a new value. This process is shown in Listing 10.4.1, in which a list of memberships are added to a team. Reactive streams are only available for entities stored locally on the device. At the time of writing, every entity besides invites provide reactive streams.

Code listing 10.4.1: Fetching a stream of a team with memberships.

```
Stream<Team> getTeam(int id) =>
      Rx.combineLatest2<Team, List<Membership>, Team>(
        _teamLocalDao.getTeam(id),
        _membershipRepository.getMemberships(teamId: id),
        (team, memberships) {
          team.members = memberships;
          return team;
        },
      );
```

10.5 Synchronization

In order to maintain consistency between the state of the local database and the remote database, synchronization is performed regularly. The application attempts to synchronize every time it is opened. Furthermore, user actions resulting in large state changes, such as leaving a team, will result in a full synchronization due to the amount of data changes.

Synchronization functionality is provided through the use of three components: The *synchronization* service, the *synchronizable* interface and the *synced_at* object. Every object implementing the synchronizable interface must provide functionality for safely upserting and deleting a set of entities within the same transaction. The *synced_at* object is responsible for keeping track of when a synchronizable entity was last synchronized with the server, and is persisted to local storage after every synchronization. These two parts are utilized by the synchronization service. It sends a request to the */synchronize* endpoint of the server and provides the *synced_at* object as query parameters. The server reads the times at which the application last synchronized, and returns all changes newer than the provided times. It will return a json object separated by entity, where each entity has a *created*, *modified*, and *deleted* field. Once the synchronization service receives a

response, it will initiate a transaction, in which it will call all synchronizable DAO's and repositories. In order for the synchronization to be considered successful, all operations within the transaction must be successful. If one fails, the synchronization is considered a failure, and any local database changes are reverted. If it succeeds, the service creates a new *synced_at* object and persists it to local storage.

Offline areas and invites are not synchronized with the server. Instead, offline areas are kept entirely local. Besides avoiding the process of re-downloading maps when switching mobile devices, there are no benefits to storing the areas on the server-side. Similar reasoning was used when deciding on invite synchronization. Invites are not required for offline usage, and provides no real benefit besides a reduction in HTTP requests towards the server.

10.6 Navigation

In Flutter, navigation is performed through the *Navigator* widget. Every *MaterialApp* has an implicit *Navigator* accessible through the build context, which is usually only available in widget build functions. Thus, in order to navigate, one would have to mix UI code with navigation logic, which in turn would have to be called from a callback in the view model. In short, it would break the separation between UI code and logic. An alternative is to utilize a *GlobalKey*, a key guaranteed to be unique across the entire application. The *MaterialApp Navigator* accepts a *GlobalKey* as a parameter. Through the use of the *GlobalKey*, one is able to programmatically navigate using the navigator it is associated with. If one were to make this *GlobalKey* global, all view models would be able to access it. This approach is not very beneficial, as navigation logic would have to be rewritten every time one has to navigate in a new view model, resulting in code duplication and implementation variance. Furthermore, usage of global variables can quickly become hard to keep track of. Finally, navigation is such an unique concern that it should be handled separately from everything else.

This application solves this problem by creating a service dedicated to navigation. It exposes a set of predefined functions one can use to navigate throughout the application, and can be injected into view models with *Provider*. The navigation service exposes a *GlobalKey*, which the *Navigator* of the root *MaterialApp* uses as its key. With the navigation service, the developer is able to programmatically navigate from one view to another through the use of named routes. Every view within the application has an associated route name. Whenever a new named route is pushed on top of the *Navigator*, it creates the view associated with the specified route name. Mapping a route name to a view is done using an application router. The router is a static switch function, in which every case handles a single route name. When the correct switch case is found, the associated view is instantiated and pushed on top of the *Navigator* stack.

10.7 Snackbars and Dialogs

Section 10.6 describes the problem of having navigation logic within UI classes. Several other aspects of Flutter faces the same challenges, albeit with much more complex solutions. Snackbars, alerts and dialogs are core concepts within many apps, and Flutter requires all of them to have access to the current build context.

A Snackbar requires a *Scaffold* in order to display itself. A Scaffold can be viewed as the base for any screen in Flutter. It provides the default layout for a Material application. Every view has their own scaffold. The approach chosen for navigation would not work, seeing as each scaffold must have a unique key. One could make every view call the "snackbar" service to register the new key, and call the service again when it is removed. This, however, seems overtly complex, and would introduce an unnecessary risk of error. Another solution would be to nest scaffolds, with one scaffold containing the entire application. This is not recommended, as nesting scaffolds can lead to undefined behavior.

Luckily, Flutter released a solution for this during the development period of this project. *ScaffoldMessengerState* allows one to define a snackbar service in the same way the navigation service is implemented. By passing it as a parameter to the *MaterialApp*, all scaffolds will be registered with this key as its "messaging key". A snackbar is a widget, and should therefore not be created in any view models. The author solved this by defining a factory for snackbar creation. The factory expects a set of snackbar specific option objects the developer can utilize to customize the snackbar. Snackbars can be programmatically shown by utilizing a set of functions provided by a service dedicated to UI interactions, encapsulating the entire creation process. This still introduces some coupling between UI and logic within the service, but at a much smaller rate.

Dialogs and alerts are implemented in the same way as the snackbars, with the view model simply needing to call the service with the function exposing the desired dialog or alert. The only difference between the two is that dialogs and alerts uses the navigation key, whereas snackbars uses the aforementioned *ScaffoldMessenger-State*. Snackbars, dialogs and alerts are used in nearly every view model. Seeing as their use cases are so similar, they are grouped in the same service to simplify view model dependencies.

10.8 Authentication

Authentication is implemented through the use of the popular *AppAuth* library, which is directly mentioned in the best practices document outlined by the IETF in RFC8252 [65]. The library is encapsulated within an authentication service, which exposes functionality for logging in, logging out and refreshing access tokens. The service is agnostic with regards to identity providers. Actual configuration details

for the identity provider are defined in a separate configuration file.

AppAuth provides functionality for automatically opening a web browser with the login page for the configured identity provider. Successful logins are automatically detected by *AppAuth*, which closes the web browser and returns the user to the application. The authentication flow results in an access token and a refresh token. Access tokens are short-lived, and are used for accessing a specific API. In order to enable offline applications, the refresh token has a much longer life-span, and can be sent to the identity provider in exchange for a new set of tokens. These tokens are stored securely on the device through the *SecureStorageService*, which will be discussed further in Section 10.11. The process of logging out is less complicated; The service simply clears all user data from the system memory and navigates the user to the initial application view.

10.9 Application Services

The application services available throughout the application are only concerned with use cases with the need to orchestrate several separates pieces of logic in a specific order. An example of this is shown in Listing 10.9.1, in which the use case of aborting the ongoing trip is achieved by orchestrating the preferences service and the trip repository.

Code listing 10.9.1: Application-layer logic for deleting an ongoing trip.

```
Future<Response> deleteOngoing() async {
    var tripId = await _preferences.readInt(PreferenceKeys.ongoingTripId);
    if (tripId == null) {
      // Trip must actually exist.
      return Response(error: "No trip is currently ongoing");
    }
    var trip = await _tripRepository.getTrip(tripId).first;
    return await _tripRepository
        .deleteTrip(trip)
        .then((value) => _preferences.delete(PreferenceKeys.ongoingTripId))
        .then((value) => _preferences.delete(PreferenceKeys.ongoingTripMapId))
        .then((value) => Response())
        .catchError(
            (err) => Response(error: "Failed to delete ongoing trip: $err"));

}
```

As the development progressed, application service usage proved to be less prevalent than expected. The development process proved that most use cases throughout the application can be achieved by calling a single repository function. Introducing a mediator would provide no immediate benefits; Instead, the author decided to follow the principle of YAGNI, and only create application services when it provided a benefit not already available through other means. If a specific use case ever grows to require more functionality, converting it to an application service would be a small task.

10.10 Location

Retrieving an accurate representation of the user location is vital in order for the application to function as intended. When running in the foreground, the application utilizes a location service for receiving a continuous stream of location updates. The service mostly acts as an encapsulation of GeoLocator [79], the framework used for accessing native location services. By default, location updates are emitted periodically, regardless of whether the user has moved or not. In order to allow for minimal location update duplicates, one can choose to pass in a distance filter, which in turn will ensure that the stream only emits location updates when a new user location is farther away from the previously emitted value than the distance provided as a filter.

Whereas this approach works well when the user is actively using the application, it stops working once the application is moved to the background, or when the phone is locked. The moment a user navigates away from the application, all running business logic is paused until the application returns to the foreground. Thus, background location tracking must be performed in an isolated thread to ensure continuous location updates regardless of the current application state. This is handled by a background location service encapsulating usage of the *background_locator* package [67]. The service is started when the application enters a paused state and stopped once the application returns to the foreground. Because the background location update functionality runs in an isolated thread, it cannot access any application functionality. Location updates are instead persisted to a file to allow for the application to retrieve them once it returns to the foreground.

10.11 Storage

In order to fulfill every requirement, the application provides services for file storage, secure storage and preference storage. The file storage service is used for storing downloaded map tiles and images taken during supervision trips. It exposes API's for retrieving, modifying, deleting and writing to files. Furthermore, it allows one to create, delete and rename entire directories. The application is only allowed to access a specific portion of the underlying platform storage. Storage allotted to the application is located through the use of the *path_provider* [92] plugin.

The domain of secure storage is solved natively on both iOS and Android; There is no need to implement it again. The *flutter_secure_storage* plugin allows access to the native encrypted storage solution of the underlying platform [77]. Usage of the plugin is encapsulated within a service, exposing API's for writing, deleting and retrieving values stored under a key name. The service only accepts a set of predefined keys as valid key values. Preference storage is implemented in the same manner. It simply acts as a wrapper around a plugin for accessing platform-specific preference implementations, reducing the cost of switching libraries.

10.12 Downloading Maps

Downloading and managing offline maps is one of the more complex domains the application handles. Functionality for downloading maps for offline use is not available in any Flutter mapping package; Thus, it had to be implemented from scratch. Luckily, the *flutter_map* [72] package allows one to read map tiles from the file system. Therefore, the author only had to implement a way to calculate, download and store map tiles in such a way that *flutter_map* can retrieve them. This section describes how maps are stored locally and how the required map tiles are calculated and downloaded.

In order to understand the decisions made in this section, one must have a rudimentary understanding of internet mapping technologies. Internet maps are oftentimes projected from a sphere shape to a flat, rectangular shape. The *Web Mercator (EPSG 3857)* projection has become the standard projection method for the web. Web maps are usually shown as a set of images, each representing a portion of an entire map. These portions are named map tiles. In order to allow for zooming in or out, map tiles are provided for a set of predefined zoom levels. Zoom level 0 usually represents the entirety of the available map. That is to say, combining every map tile available for zoom level 0 would form a complete map. As the zoom levels increase, a map tile comprises a smaller portion of the map. For instance, each additional zoom layer could halve the covered area of any map tile. The Web Mercator allows one to convert a coordinate to a specific map tile at any zoom level, and vice versa. As such, by having a set of coordinates, one is able to transform the map bounds into a set of tile coordinates.

In the application, map portions can be selected for download by dragging from one screen position to another. The start- and end positions of the drag action are stored as screen coordinates, with no association to actual map coordinates. *Flutter_map* allows one to retrieve map coordinates for the center and the boundaries of the currently displayed map portion. As such, one can convert a screen position to a map coordinate by transforming the map center coordinate into a screen position, detract the marked screen position and convert it back into a map coordinate.

Once map bounds have been defined in terms of map coordinates, all the required information for storing a map to the database is ready. However, before this can be done, one has to download the required map tiles. If one were to store the map as-is at the current zoom level, the selection would not be able to fill out the entire screen. Therefore, the application scales up the selection until it is large enough to fill the entire screen.

Once the minimum zoom level has been adapted to fit the screen size, the map can be downloaded and stored. Seeing as the bounds of the selection are already defined, this is a simple process. The north-eastern and the south-eastern selection points are converted into tile coordinates. By combining the two points, one is able

to extract the two other corners of the selection and calculate every tile contained within the calculated bounds.

The tile images are stored to the file system portion belonging to the application. Each tile will be stored as **$HOME/maps/$NAME/$Z/$X/$Y.png**, where **Z** is the zoom level, **X** and **Y** the tile coordinates, and **NAME** the map name.

Tile images are downloaded from a tile server belonging to *Kartverket* at the behest of the project supervisor. Images can be downloaded through GET requests. Throughout the development period, the author found the *Kartverket* tile server to be unreliable. At random points of the download process, a tile download would halt entirely, causing the entire download process to stop. In order to alleviate this problem, a time-out boundary of 10 seconds was introduced. If the request exceeded the boundary, it would be retried once before being considered a failure.

All logic surrounding image tile storage and downloading is contained within a service. This service exposes a stream when map downloads are being performed. The stream emits a new event every time a new map tile is downloaded, providing the data one requires to display a progress indicator in the user interface. The progress indicator dialog also allows the user to cancel the download process. A download cancellation terminates active tile downloads and deletes any locally stored tiles associated with the current map download.

10.13 Resolving Dependencies

The onion architecture is entirely dependent on Inversion of Control (IoC). For services, repositories and DAO's, the dependency resolution process is implemented with an IoC container, a lightweight container in which one can register dependency instances for retrieval later-on. The instances are created and provided using the Provider package [70]. An example of instantiating and providing a service is shown in Listing 10.13.1. Once the instance has been created, other instances depending on it will be able to reference it through Provider resolution. All instances created within the IoC container follows the dependency injection pattern. They are entirely unaware of where their dependencies are coming from.

Code listing 10.13.1: Instantiating a dependency in the IoC container.

```
Provider<INavigationService>(
  create: (context) => NavigationService(),
  dispose: (context, navigatorService) => navigatorService.dispose(),
),
```

This does not apply to view models, unfortunately. A new view model is created every time a new view is pushed onto the application navigator. Thus, dependency resolution is bound to be present whenever a view creates a new instance of its view model, resulting in views having explicit knowledge of the service locator.

This is shown in Listing 10.13.2. The downside to this approach concerns widget testing, in which the developer will have to create an IoC container with mock implementations of the dependencies the widget requires.

Code listing 10.13.2: Providing dependencies to view model.

```
return BaseView<RegisterDetailsViewModel>(
      model: RegisterDetailsViewModel(
        Provider.of(context),
        Provider.of(context),
        Provider.of(context),
      ),
      ...
);
```

10.14 Networking

A large portion of the functionality provided by the application is dependent on accessing the server, and requires network connectivity. Server communications are provided through a generic *HTTP* service, allowing one to perform POST, GET, PUT and DELETE requests towards a specific URL. Most of the available server endpoints requires the user to be authenticated. The HTTP service handles this by intercepting every outbound request in order to add an access token as an authorization header before it is sent. By appending an access token as an authorization header, every request will have all the information required by the server in order to authenticate and authorize a request. If the provided access token is expired, another interceptor is able to request a new access token and retry the request with the newly provisioned access token. If the new request also fails, the request is returned as a failure.

10.15 Documentation

The entire application is documented through the use of *dartdoc*, Dart's documentation generation tool. Whereas other tools often utilize symbols to define specific properties in their documentation, Dart opts for a simpler approach. Dartdoc comments are specified by the /// delimiter; References to parameters, variables, functions and classes are all placed within brackets, as shown in Listing 10.15.1.

Code listing 10.15.1: Example of application documentation.

```
/// Creates an instance of [OfflineMapArea] that can be persisted later-on.
/// It is primarily a helper function to keep logic out of the view model;
/// The function creates and adjusts an [OfflineMapArea] to the
/// nearest [zoom] level suited for the [selectedBounds].
OfflineMapArea createMap(
    LatLngBounds selectedBounds,
    double zoom,
    String name,
    LatLngBounds mapBounds,
);
```

Chapter 11

Application Testing and User Feedback

This chapter describes the different types of testing the mobile application underwent, and details how they were performed. Furthermore, the chapter provides details surrounding system and usability testing for the entire project, and will therefore not be covered when discussing server testing in Chapter 17.

11.1 Unit Testing

Unit tests allows one to assert that a piece of code returns an expected output when provided with a specific input. Writing unit tests for a code base allows for one to ensure the system behaves as intended, both when first creating it, and when modifying it later-on.

The mobile application is unit tested through the use of *flutter_test*, a testing framework built into Flutter. Furthermore, *Mockito* [93] is used to create mock implementations of dependencies outside the scope of the piece of code in question.

Tests are stored in a completely separate folder from the actual mobile application, aptly named "*test*". The internal folder structure of the "*test*" folder closely resembles the one described for the mobile application; The difference between the two is the existence of helper classes used for generating test data in the test folder.

All Flutter tests must have a file name ending in "*_test.dart*" in order for Flutter to register them as tests. The ending is prepended by the file name of the class being tested. If a class is stored in a file named "*widget.dart*", the test class would be named "*widget_test.dart*". An example of a unit test is shown in Listing 11.1.1, and asserts that a farm local DAO should allow for one to successfully insert several farms into the database at the same time.

Code listing 11.1.1: Unit test for the local farm DAO.

```
void main() {
 Database database;
 IFarmLocalDao farmLocalDao;

 setUp(() {
    database = Database.test();
    farmLocalDao = FarmLocalDao(database);
 });

 tearDown(() async {
    await database.close();
 });

 test('can insert several valid farms', () async {
    await farmLocalDao.upsertFarms([farm, farm2]);
    var farms = await farmLocalDao.getFarms().first;
    var farmIds = farms.map((farm) => farm.id);
    expect(farms.length, 2);
    expect(farmIds, [1, 2]);
 });
}
```

The aforementioned test also details the creation of the test database. Every unit test should, in theory, only test the code within the class in question. Testing local DAO's, on the other hand, makes little sense if one cannot ensure data is persisted to the database. The database should be recreated before every single test to ensure consistent results, a process that introduces a lot of overhead in terms of resource usage. This problem can be greatly alleviated by using in-memory databases, a quick and light-weight alternative to traditional databases. By using an in-memory database during testing, the application is able to utilize an actual database without the added performance penalty. The test database is set up in the "*setUp*" function, and destroyed in the "*tearDown*" function; Both are called before and after every test, respectively.

An example of the usage of Mockito to mock dependencies is displayed in Listing 11.1.2. The farm repository depends on a local DAO for database access, and a remote DAO for server communications. The only class being tested within this test is the farm repository. Thus, it would be beneficial to specify exactly what the aforementioned DAO's should return. After all, the unit test should only be concerned with the functionality within the farm repository. By injecting mocked instances of the DAO's into the farm repository, the developer is able to specify what each function call should return. In the case of this example, any calls to the local DAO regarding the removal of the farms with ID's 1 and 2 will be considered successful.

Code listing 11.1.2: Example unit test case for the farm repository.

```
class MockFarmRemoteDao extends Mock implements IFarmRemoteDao {}
class MockFarmLocalDao extends Mock implements IFarmLocalDao {}
void main() {
 IFarmLocalDao farmLocalDao = MockFarmLocalDao();
 IFarmRemoteDao farmRemoteDao = MockFarmRemoteDao();
 IFarmRepository farmRepository = FarmRepository(
     farmLocalDao,
     farmRemoteDao,
 );
  test('Successful farm removal should return success response', () async {
    when(farmLocalDao.removeFarms(ids: [1, 2]))
        .thenAnswer((_) => Future.value());
    var res = await farmRepository.removeFarms(ids: [1, 2]);
    expect(res.success, true);
  });
}
```

11.2 UI Testing

Whereas unit tests are concerned with the correctness of logic, UI testing is concerned with user interfaces being displayed as intended. Changes within shared UI components could potentially propagate throughout the application, and break at a certain point. Having to manually test every user interaction throughout the application is time-consuming; Thus, the author decided to utilize widget tests, a Flutter functionality allowing for automated UI testing.

Flutter provides a simplified testing environment for testing the correctness of widgets, allowing one to perform UI tests without having to instantiate the entire application. This simplified testing environment allows widget tests to be less resource intensive and time consuming. Widget tests are ran inside of a *MaterialApp*, just like the regular application. This is required in order to to allow widgets to render as they would normally. Furthermore, it is required to ensure the validity of the internationalization aspects of the application. Actual implementations of user interaction services are also made available for injection to ensure user actions will result in correct outcomes with regards to navigation, snack bars and dialogs.

Testing the views of the application automatically instantiates their corresponding view models as well. View models usually interacts with several services and repositories. Relying on actual implementations of such classes could be error-prone; Therefore, mock implementations of the dependencies are injected into the testing environment. This grants the developer full control over the test environment, allowing for one to solely focus on UI correctness.

An example widget test is shown in Listing 11.2.1. The test begins by specifying that the team service utilized by the view model should return two teams. The following line specifies the dependencies injected into the test environment, before finally specifying what widget should be rendered. *tester.pumpAndSettle()* ensures

the user interface has fully loaded before allowing the rest of the test to proceed. Finally, the last line asserts that two instances of the *TeamListItem* is present within the view.

Code listing 11.2.1: Example widget test for the Team List View.

```
class MockTeamService extends Mock implements ITeamService {}

void main() {
    ITeamService teamService = MockTeamService();

    testWidgets(
    'should contain two teams',
    (tester) async => await tester.runAsync(() async {
      when(teamService.getTeams())
          .thenAnswer((_) => Stream.value([team1, team2]));
      await tester.pumpWidget(MultiProvider(
        providers: [
          Provider<ITeamService>(create: (_) => teamService),
        ],
        child: createWidget(TeamListView()),
      ));
      await tester.pumpAndSettle();
      expect(find.byType(TeamListItem), findsNWidgets(2));
    }),
  );
}
```

11.3 System Testing

Before delivering a product, one should strive to gain an overview of whether the product fulfills the requirements it is meant to fulfill. This process is known as system testing, and usually consists of an amalgamation of different types of testing. Throughout this section, every type of test used to ensure the validity of the system will be described.

11.3.1 Testing Functional Requirements

The fulfillment of required functionality was tested by creating a set of test cases for every functional requirement listed in Section 3.4. Every test case would be predefined with expected results, and later updated to include the actual result. A functional requirement would only be considered to be valid if all test cases associated with it resulted in the expected result. Testing was mostly conducted through the use of the mobile application UI. Eligible functional requirements were also tested directly through the API UI provided by Swagger [94], a feature of the server later described in Section 16.6. In total, 72 of 79 functional requirements are fully implemented and error-free. The remaining seven functional requirements are either not implemented or only partially implemented, and are discussed further in Section 24.4.

11.3.2 Testing Non-Functional Requirements

Testing the availability of the system as a whole largely revolved around intentionally breaking parts of the system to see how the rest would react. The scenario listed in Table 3.13 surrounding the scenario of performing trips while offline was tested by simply turning all network capabilities of the test device off completely and performing a trip. The load-balancing capabilities described in Table 3.14 were tested by manually stopping a running server instance without terminating the computational instance it was running on within the cloud provider. In large, the author found no indications of a lack of availability, as the system proved to handle everything thrown at it.

Modifiability has been an ever-present focus throughout the development of the system, and has therefore been extensively tested. Constant refactors to improve the quality of the system has led the author to test the modifiability scenarios listed in Tables 3.16 and 3.17. The amount of encapsulation present throughout the server made it easy to change and extend the inner workings of a class without having the change propagate throughout the system.

The system has not been extensively penetration tested, although the author has performed rudimentary tests to ensure the security of every system component. For the most part, this revolved around attempting to modify the state of the server database and access API's without the sufficient permissions. The security scenarios listed in Tables 3.18 and 3.19 were both fulfilled, as the author could neither access the database nor modify a farm not belonging to the user he logged in as. A more structured form of penetration testing should be performed before the system is released to the general public. The rudimentary penetration testing performed throughout the system testing is not guaranteed to have covered every attack surface.

11.3.3 Usability Testing

The concept of usability testing aims to figure out whether the product in question is easy to use and understand. Usability tests are usually performed on subjects likely to use the product. However, this proved to be difficult due to the ongoing COVID-19 pandemic. As such, usability tests were instead performed on subjects convenient to the author. In total, three subjects were tested.

The usability testing performed within this project can be divided in two. Throughout the development process, the author continuously tested new functionality on several test participants, albeit in a very informal manner. The main purpose of these informal usability tests were to quickly assess the pros and cons of different approaches in order to quickly eliminate sub-par concepts. Once the application became more complete, usability tests were performed in a more formal manner. This approach will be discussed in the following paragraphs.

Before formal usability testing started, the author meticulously planned the test structure. Seeing as the test subjects were likely to have no prior knowledge of sheep supervision, a brief introduction of the problem area would introduce the test. Following this, the author provided a brief overview of usability testing, and why it was required for this project. The information provided to the test subjects all followed the script shown in Appendix A. Once the subjects had an understanding of the problem area and the concept of usability testing, they were asked to perform a set of tasks through the use of the mobile application.

Once the tasks were completed, each participant were asked to fill out a System Usability Scale (SUS): a 10 item questionnaire measuring the usability of the system [95]. Each question is scored on a scale between one and five, in which one represents "strongly disagree", whereas five represents "strongly agree". The SUS questions, along with the average score of the collected answers, are displayed in Table 11.1. Decimal values were rounded to their nearest integer. As prescribed by Brooke, the SUS test were filled out immediately after performing the aforementioned tasks. The subjects were encouraged to answer the questions without deliberation.

No.	**Question**	**1**	**2**	**3**	**4**	**5**
1	I think that I would like to use this system frequently				X	
2	I found the system unnecessarily complex		X			
3	I thought the system was easy to use				X	
4	I think that I would need the support of a technical person to be able to use this system	X				
5	I found the various functions in this system were well integrated				X	
6	I thought there was too much inconsistency in this system		X			
7	I would imagine that most people would learn to use this system very quickly					X
8	I found the system very cumbersome to use	X				
9	I felt very confident using the system				X	
10	I needed to learn a lot of things before I could get going with this system	X				

Table 11.1: Average results of the SUS questionnaire answers.

Brooke also provided a way to produce a singular number result representing the overall usability of the product. This is performed by transforming every response value, adding them all together and multiplying the end sum by 2.5. The transformation an answer goes through depends on their placement within the SUS form. Responses for questions 1, 3, 5, 7, and 9 are transformed by detracting one from their initial response, whereas responses to every other question is transformed by detracting the initial response from 5 [95].

The overall SUS score for the usability tests performed within this project was 83 out of 100, indicating a good user experience. However, some improvements were still present.

A reoccurring theme throughout the usability tests related to the deletion of different items throughout the application. Some subjects completely ignored "remove" buttons and immediately attempted to remove items by swiping, whereas others reacted in the opposite manner. When asked, the subjects simply explained their behavior to be habitual.

Another subject pointed out that managing the selection reticle for observation selection was hard due to their thumb overlapping it. The author noted this, and immediately changed it to instead be displayed above the thumb of the subject.

A subject also showed uncertainty surrounding the process of registering sheep ties through the use of swiping, claiming that they were uncertain of whether one should swipe towards or away from the color they wanted to register. In order to rectify this, the author added a description in the middle of the view, describing that one should swipe towards the color one wished to register.

In large, the usability testing appeared to prove the user-friendliness of the system. The usability scenario listed in Table 3.21 appeared to have been fulfilled, as the test subjects were able to comfortably navigate the application well within the 30 minute time constraint. Cancellation of map downloads, as prescribed by the scenario in Table 3.20, also proved to be intuitive for every user.

Chapter 12

Application Discussion

The development of the Flutter application took place between September 2020 and May 2021. Within this time span, both Flutter and the experience the author had with the framework grew significantly. This section will discuss several key aspects of the development process, and the challenges the author faced throughout it.

12.1 Growing pains

Flutter was released in May of 2017. Since then, the framework has grown massively. In March of 2021, the framework received a new major version, 2.0 [96]. This version had the goal of stabilizing the Flutter API: Breaking changes would be less frequent. The upgrade process proved to be successful, with the entire process requiring few changes within the application. This could be attributed to the fact that the author worked on a beta version of Flutter throughout most of the SDLC. The application required snack bars to remain on screen even after navigating from one view to another, something which was not possible on the stable release at the time of development. Thus, the beta channel had to be used. As of Flutter 2.0, this is no longer required, as the aforementioned snack bar functionality has been added to the stable release. It is important to note that Flutter still has several flaws; The growing pains the framework experiences can be very noticeable at times.

A good example of these growing pains is the performance of iOS animations. In short, iOS animations are susceptible to lower frame-rates the first time an animation is shown, and occurs every time the application is opened. At the time of writing, the Flutter team are unsure of how the issue can be resolved [97]. Although such issues do not affect this application significantly, they do raise concerns with regards to the longevity of the framework.

A more critical issue surrounds the "official" Flutter packages. Google provides several business-critical packages for Flutter. However, many of them are in various states of disrepair. An example of this is the official camera plugin, an essential functionality for many applications. During development, the package did not provide crucial camera functionality, leaving the author to opt for a third-party solution. The official plugin has since improved, and is now a viable alternative. Nonetheless, such growing pains are apparent after working with the framework for some time.

In retrospect, the author still believes Flutter to have been a good choice. Even though the aforementioned issues are a cause for concern, they do not invalidate the framework as a whole. The author still believes the productivity increase from creating user interfaces in Flutter is unparalleled, and plays a large part in the success of the project as a whole.

12.2 Problematic Third-party Libraries

In order for Flutter to be a viable solution, it requires an active developer community surrounding it. Many of the business critical functionalities within this application utilizes third-party packages where Flutter does not provide official solutions. This was known when first choosing Flutter, and has for the most part been a pain-free experience. Packages have proven to be well-documented and applicable to the problems this application had to solve. However, some problems did present themselves, with the most apparent ones relating to location and permissions.

Throughout the development period, the application experimented with a plethora of packages for requesting user permissions and acquiring the user position. At one point during the development, the location library the application utilized completely stopped emitting location updates. If this issue would not be resolved, the author would have to implement native location implementations for both platforms. This would have led to a considerable loss in valuable development time. Luckily, the author was able to resolve the problem by switching to another package and upgrading Flutter. At another point, location permission prompts stopped functioning entirely. These occurrences are rare, but not unheard of. If such business-critical packages were to be actively maintained by Flutter staff, the framework could be considered to be more stable. Relying on volunteer work on open-source solutions is risky, especially if the package is not overtly popular.

These problems taught the author a valuable development lesson: Always encapsulate business-critical dependencies in such a way that they can be easily replaced. Having a change within a package ripple through the entire project is a major inconvenience; Changing the implementation of a service is a minor change.

12.3 Over-Engineering

Some of the focal points of the application are modifiability and extensibility. The entire application has loose coupling, with the majority of implementation details being hidden behind interfaces. Adding, changing or entirely removing a DAO, for example, can be achieved without the rest of the application having any knowledge of the change. Furthermore, internationalization, dynamic theming and resizing are all supported.

The focus on extensibility and modifiability also bled into the application architecture. At the start of the project, the application architecture was much simpler than it currently is. It evolved from being a simple MVVM architecture into a layered architecture, before finally being changed to the implementation of the Onion architecture it is today. One could argue that the Onion Architecture is unnecessary for this application; The lack of concrete business rules removes a lot of the value brought by the Onion Architecture. Furthermore, application services are not used to the extent they usually are throughout most other Onion architectures. Many view models directly references repositories, as no additional logic is required to fulfill a use case. The author believes these to be valid points, yet he chose to utilize the onion architecture nonetheless, under the assumption that it would be beneficial if the application were to be developed further and placed into a production environment. Essential architectural choices should be made early; Introducing such abrupt changes late in the development process could result in an unmanageable amount of technical debt. Ensuring the architecture is able to handle modification and extensions in the future is considered crucial. Thus, the author believes the current architecture to be a good fit.

12.4 Development Experience

After having spent the better part of a year experimenting with Flutter, the author is left with an impressive opinion of the framework. The way UI is declared, combined with the predefined widgets Flutter provides, makes UI development into a trivial process. Having previous experiences with both React Native and Xamarin, the author strongly believes Flutter increases developer productivity more than the alternatives. The Dart language is also quite interesting. It is designed to be instantly recognizable for developers with previous programming experiences. In many ways, the language syntax can be compared to Java or C#. Being able to write production-ready code within the first week of picking up the framework is an impressive feat.

Although Dart can feel familiar to most developers with an object-oriented background, it does lack some functionality. After all, it is still a young language. Features like data classes and static reflection have still not been fully added. Null safety was added to both Dart and Flutter throughout the development pro-

cess, something which the author has not been able to migrate to yet, due to business-critical packages not being migration-ready.

The tooling provided by Flutter also proved to be of great help. Hot restarting and reloading greatly increased productivity. Being able to instantly experience changes in code without having to restart the application is a welcome optimization. There is almost never a need to restart the application entirely, something which the other frameworks have failed to deliver on in previous projects the author has experiences from.

12.5 Tie Registration

As previously stated in Section 3.4, the observer must be able to conduct sheep tie registration without having to look at the screen. This proved to the most challenging user interface design to plan throughout the entire development process. In order to solve the problem, several prototypes had to be created. The first area the author explored was speech-to-text; If the observer could simply specify a color and an amount by voice, the entire problem would be solved. However, such a solution is susceptible to background noise. Windy days, sheep bells and other natural causes of sound could make such a solution cumbersome to use. As such, the idea would be abandoned.

Further exploration led to the process of registering details by swiping. The process of swiping is natural to every smartphone user, and does not necessarily require one to look at the screen. Within the area of swiping, two different solutions were explored.

A smartphone screen has four sides, perfectly corresponding to the number of tie colors present in the NSG standard. If one were to allot one color to each side of the screen, the observer could simply swipe in the direction of the desired color. Such an approach would require the observer to remember what side of the screen each color is allotted to. The association between color and swipe direction could be strengthened by having the entire swipe-to-register process in a dedicated view, allowing one to spread the colors across the entire screen. Furthermore, text-to-speech could be utilized to continuously tell the user what number has been specified for any color.

The other swipe approach also utilizes all sides of the screen, although in an entirely different way. By devoting the entire screen to a specific color, the observer could swipe up or down to change the current color to the next or previous color. Swiping right and left could add or detract from the color currently shown on screen. This approach would rely on an additional way to make the user aware of what color is currently selected. The most obvious solution is to inform the user of what color is currently selected through text-to-speech functionality triggered by a change in color.

Both swipe approaches can be considered to be viable solutions. Usability testing showed that the subjects preferred the first swipe solution over the other, as they did not have to rely on the text-to-speech functionality to know what color was currently selected. Although the second solution provides the user with the ability to detract from the registered number for a specific color, the process of continuously changing colors were deemed to be less user-friendly than the first solution.

12.6 Application Identity and Platform Adaptability

A major choice with regards to the design of the application was that of choosing whether or not to commit to using platform-specific UI components. Flutter provides different kinds of widgets intended for different operating systems. Swapping widgets depending on the underlying user interface requires additional work. The author decided to implement core UI components as adaptable widgets, whereas others would have a custom design. By following this approach, the application would be able to have an unique identity, while still conforming to the guidelines outlined for both Android and iOS. Adaptable components would be provided for alerts and dialogs, whereas everything else would be platform-independent.

The application features a "home" view, acting as the central hub for the entire application. Earlier revisions instead utilized bars at the top and bottom of every screen, allowing for tab navigation. An excerpt of this UI is shown in Figure 12.1.

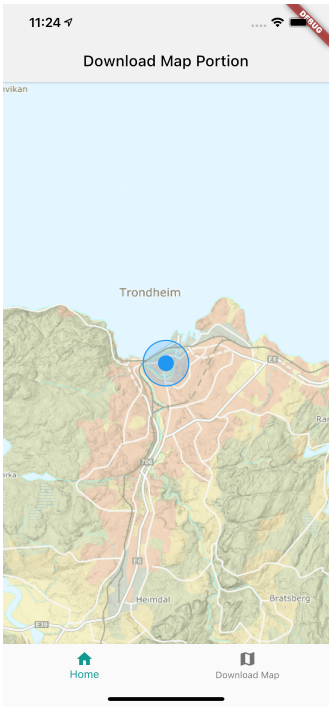

Figure 12.1: First version of application user interface.

Switching between tabs is a common feature in many applications; In the case of this application, one does not need to switch between different contexts. If the user is conducting a trip, there is no need to manage teams, view previous trips or log out. Furthermore, the navigation and application bars takes up valuable screen real estate; Moving them would allow the map to take up more screen space. Finally, only utilizing a single stack of views greatly reduced the complexity of the application, as the support for managing several navigation stacks proved to be lacking at the time the author decided to implement it. With the arrival of Flutter 2.0, support for tab-based navigation appears to have improved. However, the author still believes that utilizing tabs would provide no additional benefits to this application. The different use cases are too specialized to gain any benefit from quickly switching contexts.

12.7 Usability Tactics Usage and Deletion Consistency

The author believes that usability tactics could have played a larger part in the application, especially with regards to undoing specific user actions. Large, consequential tasks within the application utilize confirmation dialogs to ensure the user actually wishes to perform the action. For such tasks, the "undo" tactic can be considered inconsequential. Smaller actions, such as handling invites, provide no confirmation dialogs. These cases could have benefited from the "undo" tactic to account for any potential user errors. The tactic was removed due to a lack of time; Exploring it further is an avenue worth considering if the application is developed further.

At the time of writing, how an entity is deleted within the application differs drastically from item to item. In some situations, items can be deleted from a list through the aggregation tactic, in which several items can be selected and deleted in the same action. Other situations allows the deletion of items through swiping or tapping. The author fears that this inconsistency can lead to confusion among potential users. A wiser approach could possibly be to standardize the deletion method across the entire application. Choosing one over the other is a difficult decision. During usability testing, the test subjects appeared to have differing views on how items should be deleted, with some favoring singular deletion whereas other favored aggregation. None, however, mentioned the inconsistency between deletion methods. As such, the author has decided to leave it as is until a more conclusive answer to the deletion tactic debate is uncovered.

Part III

Server

Chapter 13

Server Introduction

The focal point of this system is the mobile application; It is the piece that allows sheep observers to switch from analog solutions to a digital one. If one were to only store observation data directly on the device, the overall usefulness of the system would be severely reduced. To allow for a greater level of usefulness, the data generated through supervision trips are centrally stored on a server, which will be described throughout this part. The part will go on to describe the technologies the server and database utilizes, important implementation details and how the server is tested. Finally, it will describe how the server is deployed, before concluding the part with a discussion surrounding the development of the server.

Chapter 14

Server Technologies

There are usually no clear-cut answer to choosing a technology stack, and this project is no exception. The server would have to be developed at a more rapid rate than the mobile application due to time constraints; Technologies allowing for rapid and scalable development could therefore prove to be beneficial. This chapter will describe the different technologies the server utilizes, and reason as to why they were chosen.

14.1 Database Management System

As previously discussed in Section 10.2, the data generated by the system is stored in relational databases. The section goes on to describe the usage of SQLite [98], a light-weight Relational Database Management System (RDBMS) perfect for use in mobile applications. At larger scales, however, other alternatives are more capable. The server utilizes PostgreSQL, a free and open source RDBMS boasting a near full compliance with the SQL standard [99].

In terms of performance, most RDBMS are on equal terms. In large, the performance gains or losses are nearly irrelevant for systems of this scale. Functionality, on the other hand, is of the utmost importance. At the time of writing, the system does not make use of many complex SQL functionalities. However, the ability to utilize complex functionalities later-on without having to migrate would be very beneficial. Furthermore, PostgreSQL provides a set of extensions introducing additional functionality. Seeing as the system generates a lot of geospatial data, the ability to analyze it further could be of interest as the system grows. Utilizing PostgreSQL would allow for such functionality to easily be added.

Finally, PostgreSQL is entirely free, and supported by nearly every cloud vendor. Whereas large competitors, such as Oracle and Microsoft, charges licensing fees for scaling, PostgreSQL will always be free. One should mention that this is irrelevant if

the application will be hosted through a cloud vendor, as the pricing model changes to pay-as-you-go. However, choosing a fully free RDBMS like PostgreSQL provides one with the ability to choose to self-host at a later point. This same argument can be made for MariaDB, a fully free and open-source fork of MySQL [100]. However, PostgreSQL remained the preferred choice due to the author wanting to keep the possibility of utilizing advanced functionality not yet present in MariaDB. One could also make a case for choosing cloud-vendor specific RDBMS. However, the author wanted to remain as cloud-agnostic as possible, and therefore chose to utilize PostgreSQL due to it being supported nearly everywhere.

14.2 Docker

Docker is a service that allows for one to run isolated applications through the use of virtualization [101]. By utilizing Docker, developers are able to ensure consistent environments regardless of the underlying operating system and environment. Docker has been used extensively throughout the server development; Both the server and local PostgreSQL test databases are ran through Docker. This is achieved through the use of Docker Compose, which Docker describes as "... a tool for defining and running multi-container Docker applications" [102]. Docker Compose allows one to specify what resources Docker should provide. Furthermore, it allows one to start and stop resource provisioning through the use of the command line.

14.3 Programming Language

Choosing the programming language of a project is a difficult decision to make. The programming language itself is oftentimes not the most important factor; Instead, one should decide what specific requirements the project has, and whether a language provides sufficient support for it. In the case of the server, the single most important requirement proved to be at what speed one could develop in. The allotted time for server development was lesser than what was allotted for the mobile application. Languages that would allow the author to develop quickly, yet still in a scalable manner, could prove to be a good option.

The need for a quick development period removed several alternatives. Choosing a language the author already had experience in would allow for a shorter ramp-up time; Being effective from the get-go is invaluable. The following languages were considered when planning the server:

- JavaScript (Node.js)
- Python
- Java
- Kotlin
- Dart

The mobile application is developed through the use of Flutter, which utilizes Dart. Being able to utilize the same programming language throughout the system would decrease the knowledge gap between any future frontend and backend developers if the project were to be continued. However, it is rarely used for backend purposes. Mature and efficient web application frameworks are non-existent. Choosing Dart would leave the author with the task of developing solutions for problems the other languages already solved. As such, Dart proved to be a bad fit for the project.

In the same vein of developer productivity maximization, the author decided to remove Java as an alternative. Kotlin is a programming language built upon the Java virtual machine. The language has achieved a meteoric rise in popularity, with many choosing to abandon Java for it. It is less verbose, allowing for developers to write less code. Kotlin even allows for Java and Kotlin interoperation, meaning that any Java features are available in Kotlin. Furthermore, it both addresses current issues with Java as a language, as well as adding several improvements and new concepts, as listed in [103]. One can certainly make a case for choosing Java over Kotlin, but the author believed that Kotlin would be the better choice of the two in the context of this application, as it would reduce the development time.

14.3.1 Choosing JavaScript

Having eliminated both Java and Dart, the author chose to utilize JavaScript. The main motivator behind the choice is developer experience. Of the three remaining options, the author feels most productive when using JavaScript. In the modern age, development costs are much higher than the cost of computational power; Even if Python or Kotlin proved to be more performant, the author believed that the boost in productivity would be worth it. The application will never scale to a size in which language performance will cause a considerable impact. Furthermore, the author prefers to utilize languages with support for static typing, a feature Python does not provide. As such, JavaScript felt like the most natural choice.

JavaScript has traditionally been utilized to perform scripting on websites. However, as time went on, the language has grown to accommodate several other runtimes as well. One of these run-times are Node.js [104], a runtime environment allowing JavaScript to be ran outside of a web browser.

JavaScript is dynamically typed; The type of a variable is not known until the system is running and ready to interpret it. The author is of the firm belief that static type checking improves the quality of code, a sentiment reflected in previous research [105] [106]. In recent years, supersets of JavaScript supporting static typing have emerged, with TypeScript [107] being a popular example. The language allows for seamless transpilation between itself and JavaScript, allowing for the benefits of static typing while still being in the realm of JavaScript. The choice of JavaScript came with the caveat that TypeScript had to be used to allow for static typing. Had it not been for TypeScript, the project would have utilized Kotlin.

14.4 JavaScript Packages

The JavaScript environment provides a plethora of packages expediting the development process. This section will briefly describe the most essential packages the server utilizes to speed up, improve or simplify the development process.

14.4.1 Nest

The choice of web application framework usually shapes the final implementation of any project to a considerable extent. Within the Node.js ecosystem, *express* [108] is the most popular web application framework. The framework describes itself as minimalistic and unopinionated, allowing the developer to utilize it in a plethora of ways. However, as previously stated, the server had to be developed under strict time constraints. As such, minimalism would not be a good fit. Instead, the author chose to utilize *Nest.js*, a feature-rich web application framework built on top of *express* [109]. Whereas *express* is very lenient, *Nest* provides the developer with a set of tools and features that encourages a very specific architecture. This architecture is later discussed in Chapter 15.

14.4.2 TypeORM

TypeORM [110] is an Object-Relational-Mapper (ORM) for TypeScript, and abstracts away the process of mutating the database. The framework allows one to define database schemas through the use of code. Furthermore, it allows for one to perform type-safe database mutations and queries specific to the object one is operating on. Within the ORM space of Node.js, TypeORM appeared to be the most mature option. Furthermore, it covered server requirements other ORM's did not.

14.4.3 Axios

Machine-to-machine interactions are handled through the use of Axios [111], a HTTP client providing simplified HTTP functionality over the default functionality provided by Node. The library was chosen due to the ease of which one can intercept requests and responses before they are sent and received, respectively. It is only used for interacting with the identity provider the system utilizes.

14.4.4 AWS SDK

The majority of the cloud infrastructure of this system is hosted on AWS. In order to effectively communicate with provisioned AWS resources, Amazon created the AWS SDK, a set of API's allowing the developer to perform actions related to a resource directly from JavaScript [112]. The server uses the library to store images to an S3 bucket, and is discussed further in Section 16.13.

14.4.5 Node-cache

The system utilizes Auth0 to store user credentials. Auth0 allows one to retrieve specific user information through the use of a management API, which the server utilizes to retrieve and manage users. The retrieval process is largely mitigated through the use of caching; Every time a new user is fetched from Auth0, it is stored in an in-memory cache provided by *Node-cache* [113]. By caching the user information, future requests can be fulfilled at a greater speed by eliminating the round-trip to Auth0.

14.4.6 Passport

When creating secure applications, authentication is a must. This server utilizes the *Passport* [114] framework to authenticate every request that mutates the database. The framework serves as middleware, a piece of functionality with the choice of either handling, transforming or forwarding a request before it is handled. Before every request is handled, the server utilizes the *Passport-jwt* strategy to verify the validity of the JWT identifier included in the request.

14.4.7 ESLint

ESLint [115] allows the developer to ensure that the project is following the best available development practices by introducing a set of static analysis tools. Furthermore, ESLint provides the developer with tooltips and automatic code changes whenever a known error is encountered. By utilizing ESLint, the developer not only ensures the quality of the code, but also a consistent style throughout the entire project.

14.4.8 Prettier

Prettier is a tool that enables consistent code formatting throughout a project [116]. In this project, Prettier is utilized in such a way that code is automatically formatted to conform to a specific style whenever a file is modified and saved. ESLint and Prettier are configured to work together, allowing one to conform to the rules outlined by both Prettier and ESLint without conflict.

Chapter 15

Server Architecture

The architecture of the server largely reflects the mobile application architecture discussed in Chapter 9. They do, however, serve drastically different purposes. The mobile application is tailored towards performing a specific subset of tasks required for digital sheep supervision, and therefore has a more defined set of requirements than the server. At the time of writing, the server mostly handles CRUD concerns. However, ensuring that it can be expanded upon to include additional functionality later-on is of the utmost importance. As with the mobile application, the author believed that utilizing the Onion architecture would prove to be the most scalable solution. The Onion architectural pattern is described in detail in Section 9.1.1, and will therefore not be detailed here. In short, the pattern separates code in layers that are only allowed to access layers closer to the core of the application than the current layer. A correct implementation of the pattern allows for each layer to be completely unaware of the technology used within the other layers. Seeing as the server utilizes a plethora of packages, ensuring that one can be easily swapped out without causing a rippling effect throughout the system is important.

15.1 Server Design Patterns

The following subsections will describe the different design patterns used throughout the server, and how they are utilized.

15.1.1 Data Transfer Object

Fowler describes a Data Transfer Object (DTO) as "an object that carries data between processes in order to reduce the number of methods calls" [117]. DTO's are usually capable of serialization and deserializing its content, making them perfect for receiving and transmitting data in a web server. Most of the server endpoints only accepts data that can be deserialized into a predefined DTO.

15.1.2 Chain of Responsibility

Every request the server handles goes through a variable amount of steps before being processed; User input must be validated, and requests must be authorized. Performing several of the aforementioned checks for every request can quickly become difficult to maintain or extend. The Chain of Responsibility pattern handles this by allocating such tasks to a handler, an object that can either choose to forward, alter or handle a request [82]. Handlers are linked together in a chain that ends when a handler decides to handle the request instead of forwarding it. In this server, the chain of responsibility consists of a set of middleware; Functionality performed before a request is handled by the actual endpoint handler. Using the Chain of Responsibility pattern can simplify the addition or removal of handlers.

15.1.3 Dependency Injection

As previously stated in Section 9.2.2, dependency injection aims to achieve Inversion of Control. In short, the pattern separates instantiation and usage of objects, allowing for a testable system with low coupling [83]. Dependency injection is an integral part of Nest, and is used throughout the server to inject resources.

15.1.4 Repository

The repository pattern is already described in Section 9.2.4, and will not be described here. Usage of the pattern largely correlates to the mobile application usage: It serves as an abstraction for aggregate root persistence. Whereas the mobile application repositories utilizes dedicated DAO's to mutate the database, the server repositories utilizes an ORM.

15.1.5 Mediator

As stated in Section 9.2.9, the mediator pattern is used to orchestrate a set of objects to achieve a specific goal. Usage of the pattern within the server corresponds to its usage in the mobile application: Application services orchestrates the order in which a set of objects are called to remove any dependencies between them.

15.1.6 Template Method

Performing CRUD operations on a set of database tables through a repository patterns oftentimes requires one to create functionality only differing from one another in a single step. Gamme et al. suggests the usage of the Template Method pattern to solve such problems [82]. The pattern describes the creation of an abstract super-class specifying all non-varying steps, and defers the varying steps to implementation classes. This not only reduces code duplication, but also allows for one to easily change the behavior of all sub-classes. Every repository within the server extends an abstract repository class, and only specifies a certain amount of properties to modify the super repository to fit the targeted database entity.

15.2 Server Project Structure

As stated in Section 9.3, project structure often corresponds to the chosen architectural patterns. Both the server and the mobile application utilizes the Onion pattern, but are structured in different ways. Nest requires the developer to separate domains into specific modules that can be utilized throughout the system, and is required in order to enable the dependency injection functionality provided by the framework. Every module is stored in their own folder, which in turn is separated into sub-folders representing layers. The available packages within the system are shown in Figure 15.1. Packages are not actually separated by layer; The figure simply chooses to display them as such in order to showcase the different packages utilized within every layer. Furthermore, cross-layer usage has been generalized to the arrows shown between layers in order to increase readability. The following paragraphs will describe the folders available within any single layer.

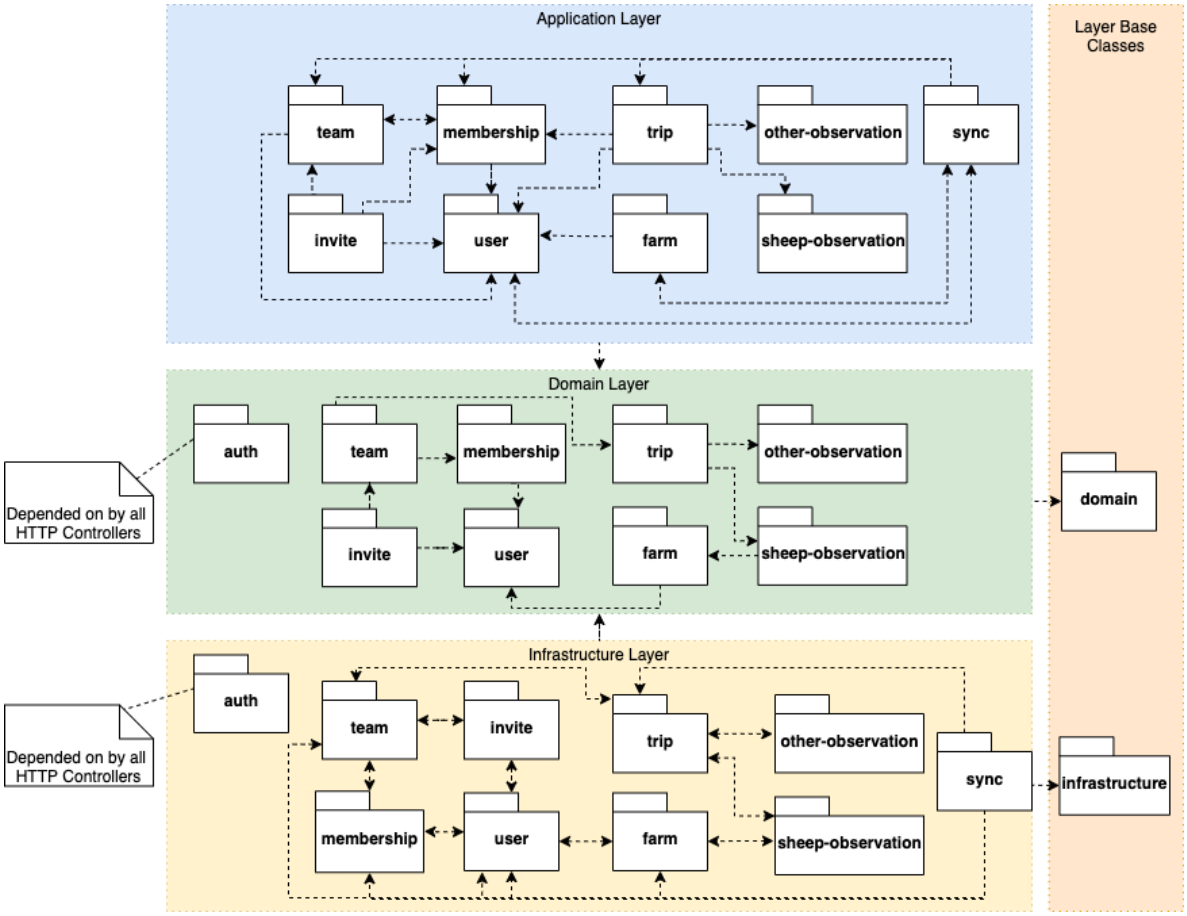

Figure 15.1: Package diagram of server architecture.

The *common* folder is home to anything not fit for a singular domain or layer. For the most part, it contains base classes that one or several objects within every domain extends. It contains the following folders:

- **domain:** Base classes for entities and repository interfaces.
- **extension:** Extensions for TypeScript classes.
- **infrastructure:** Base implementations of repositories, mappers, ORM entities and HTTP responses. It also contains database configuration files, exception interceptors and HTTP clients.

The innermost layer of the onion is *domain*. It is home to entities, domain models, business logic and interfaces defining repositories and services. Following the *domain* layer is the *application* layer, in which services used for mutating and accessing entities are stored. Furthermore, the *application* layer is home to a set of Data Transfer Object's used for defining what can and cannot be passed into a repository.

The final layer within the server architecture is the *infrastructure* layer. In terms of code organization, the layer is split in two: Code relating to the database and code relating to handling HTTP requests. HTTP folders contains a single HTTP controller and a set of DTO's documented and tailored towards the web framework the HTTP controller utilizes. The database folder consists of an ORM entity definition, an entity-specific mapper and the repository used for mutating and accessing the entity the folder relates to.

15.3 Description of Server Architecture

Figure 15.2 displays a class diagram consisting of classes related to entity persistence. The figure only displays domain classes related to the infrastructure layer, and is separated into domain-specific and common classes. Common classes usually serve as abstract base classes extended by domain-specific implementations. *IBaseRepository* defines a set of functions every repository must support, whereas *IEntityRepository* defines entity-specific repository functionality. Most repository functionality is defined in the abstract *BaseRepository* class, which every *EntityRepository* extends. The usage of an ORM allows for one to create generic functionality for persisting, finding and deleting entities. This is largely achieved through the use of the *template* pattern, allowing base classes to delegate entity-specific functionality to the concrete implementation of the class. Entity-specific repositories will only have to implement the functionality specified in *IEntityRepository*, in addition to functionality for specifying entity-specific query parameters.

OrmEntity serves as a representation of a single database table and any relations it might have. It extends *BaseOrmEntity*, which defines a set of properties shared by every ORM entity throughout the server. In order for a domain entity to be

persisted to the database, it must first be converted to an ORM entity. Mapping between the two entity types are handled automatically within the *BaseRepository* through the use of an abstract *BaseOrmMapper*. The class provides functionality for mapping between base entities; Mapping between entity-specific properties is handled by an *EntityMapper*, a concrete implementation of the template functions defined in *BaseOrmMapper*.

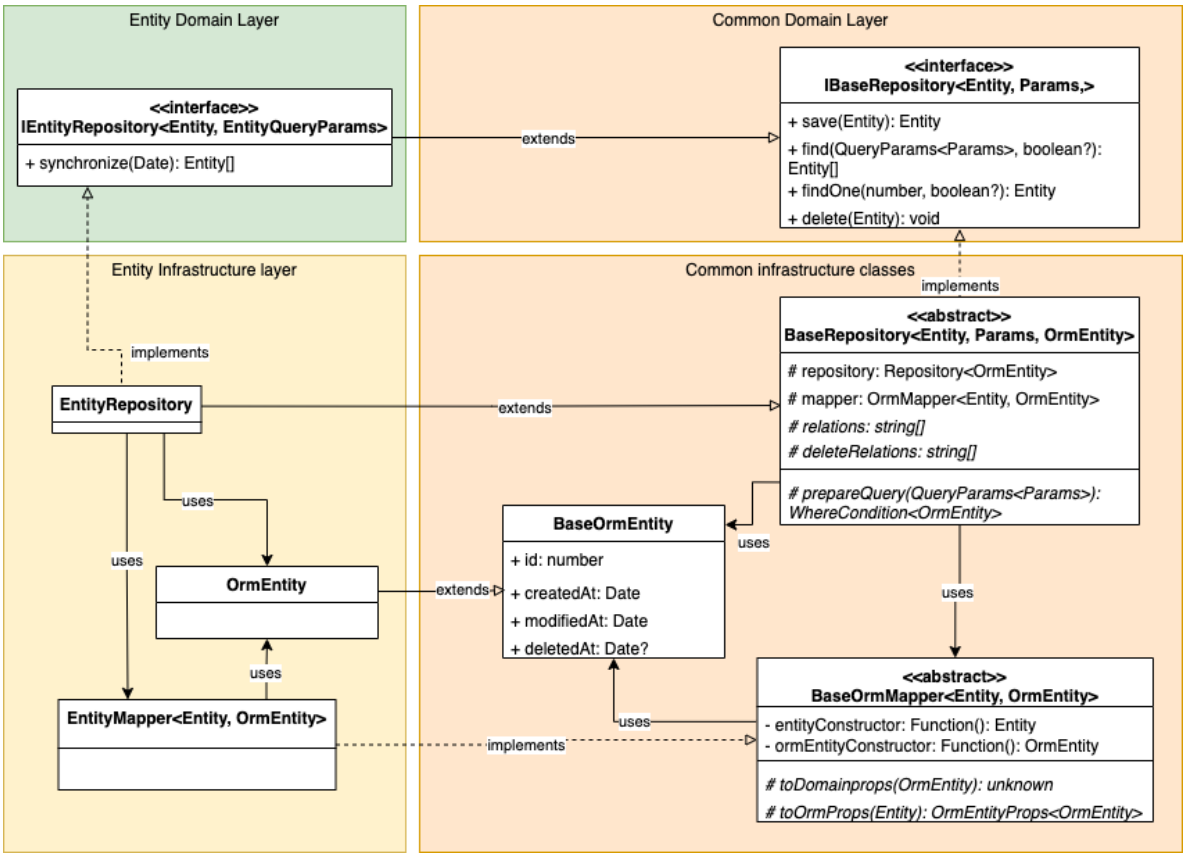

Figure 15.2: Class diagram of classes relating to entity persistence.

Figure 15.3 displays the relations between HTTP Controllers, application services, entities and repositories. The *EntityController* is responsible for receiving and responding to HTTP requests. Requests are received, parsed and validated using dedicated entity DTO's. Every DTO within the *infrastructure* layer implements a corresponding DTO interface from the *application* layer. DTO's stored within the *application* layer are technology-agnostic, and are only used to define the properties required in order to perform a specific action. This separation allows for one to completely encapsulate any HTTP-related DTO details within the infrastructure layer, allowing for one to replace the method of how to receive data without having to modify the *application* layer. If one were to change the input type from HTTP requests to CLI commands, the application layer would not be affected.

HTTP controllers make use of an application service dedicated to mutating and accessing a single underlying domain entity and a corresponding ORM entity. Every application service is only accessed through the interface it implements, allowing for one to easily change the underlying implementation if need be. The entity service will perform a set of operations to ensure the requester is authorized to perform the intended action. Furthermore, it orchestrates a set of other services in order to achieve the goal. A function within an entity service usually results in a call to an entity repository, which once again is hidden behind the interface it implements. The repository interface also specifies the two generic types *IRepositoryBase* requires. More specifically, it specifies the entity the repository handles and *EntityQueryParams*: A set of parameters the repository should filter query results by.

Every entity extends the abstract *BaseEntity* class, which contains a set of properties and functions every entity throughout the server shares. The actual implementation of an entity is represented by the *Entity* class, a generic class expecting a set of entity-specific properties as its type. The separation between the base of an entity and the actual entity properties allows for one to reduce the amount of boilerplate code present throughout the server.

The server never returns entities as a HTTP response; Instead, they are transformed into an *EntityResponse*: An object used to whitelist the entity attributes to include in a response. *EntityResponse* extends a base class that specifies properties every entity shares, allowing for less boilerplate code.

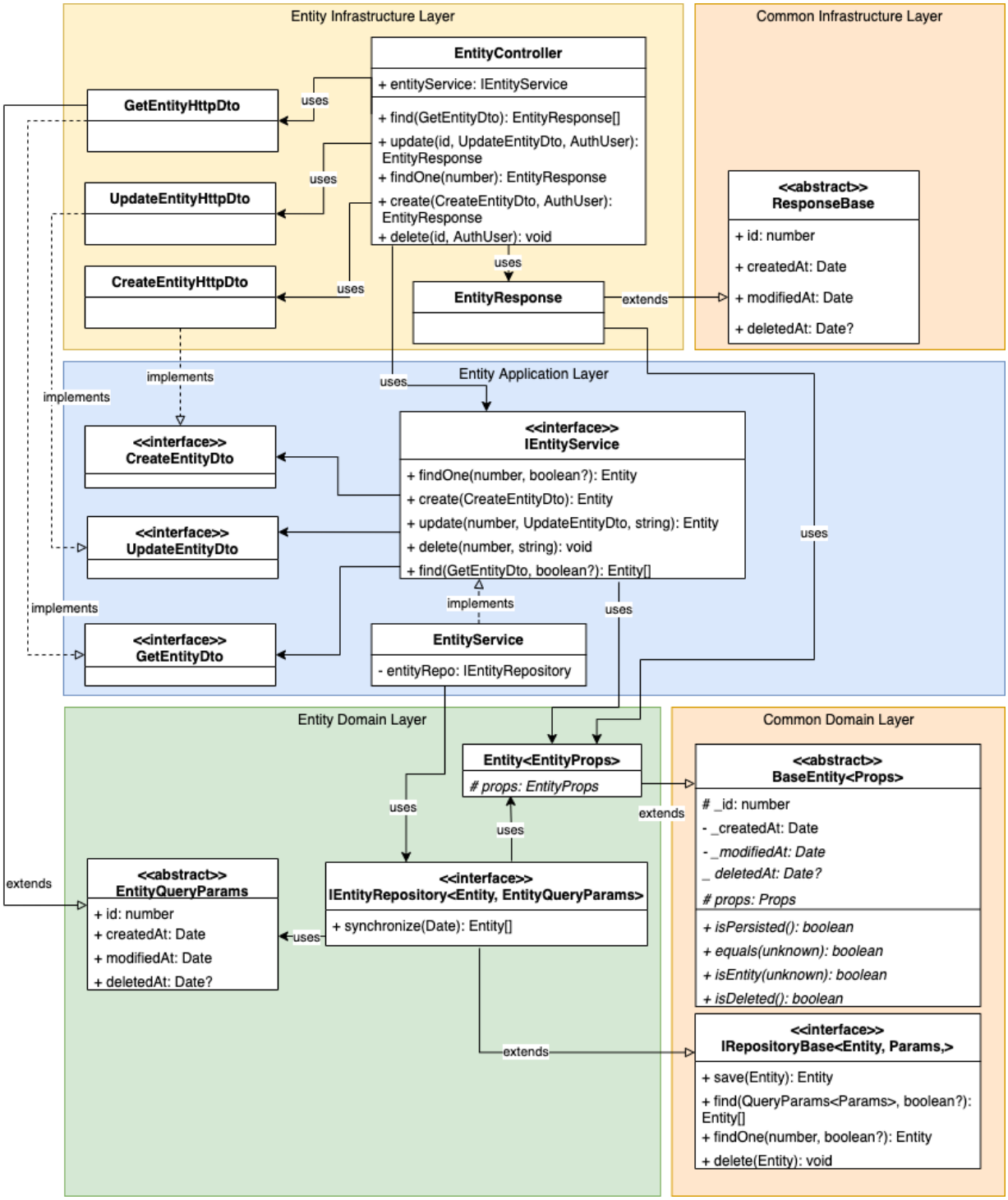

Figure 15.3: Generic class diagram describing relations between HTTP controllers, DAO's, the *application* layer and the *domain* layer.

A generic POST request is shown in Figure 15.4. In order to keep the figure at a maintainable size, it only displays the happy path. Early returns are instead described textually. The request starts when the HTTP Controller receives a new request. Once received, the controller attempts to transform and validate the request body into a predefined DTO. The controller returns an error response if the response is invalid. Requests that will mutate the database state must be authenticated through a JWT. If a JWT is attached to the request, it is sent to the identity provider for validation. Invalid JWT's will result in the controller returning an error response. Once validated, the DTO is forwarded to the entity application service, which will check if the requester is permitted to perform the intended action. Unauthorized requests will be returned as error responses. If the request is authorized, the DTO is converted into a domain entity and sent to the entity repository. The repository converts the domain entity into an ORM entity and persists it to the database. Once persisted, the repository converts the ORM entity back into a domain entity and returns it. Before responding to the request, the HTTP controller transforms the domain entity into a response object only containing a subset of the entity properties.

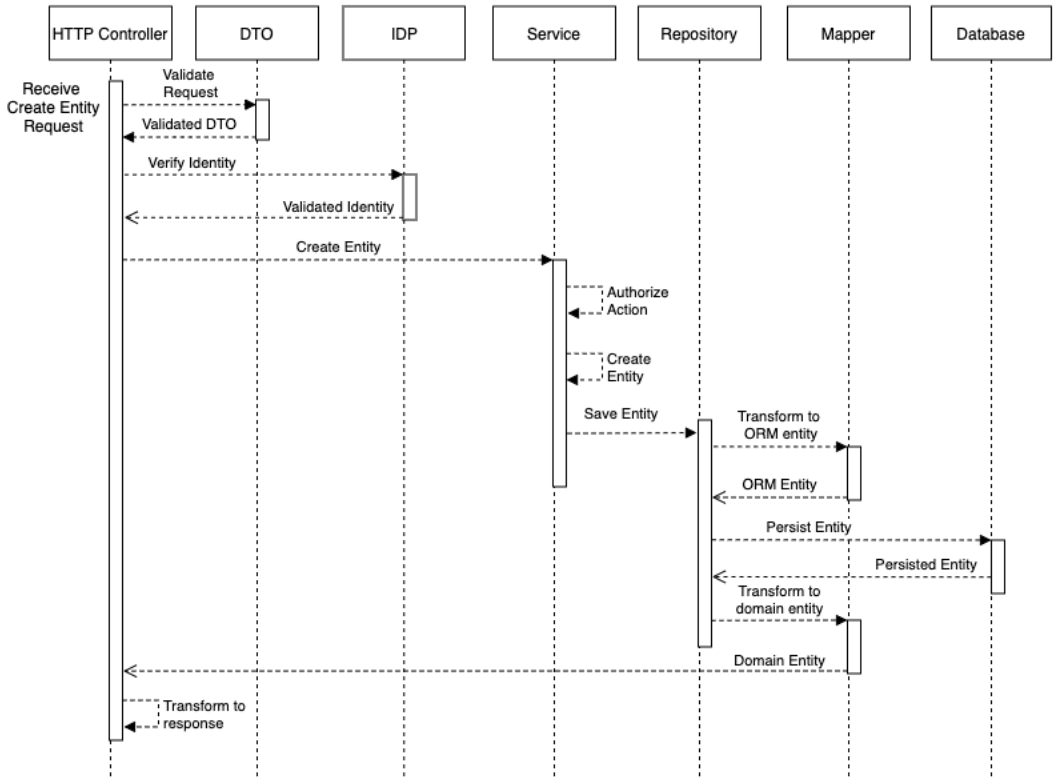

Figure 15.4: Sequence diagram describing object communications in the happy path of creating an entity.

Chapter 16

Server Implementation

This chapter will describe how the core concepts of the server is implemented. The implementation of the Onion architecture found within the server is largely based on a TypeScript implementation of the best practices surrounding the Hexagonal architecture, a pattern with many similarities to the Onion architecture. It can be found at https://github.com/Sairyss/domain-driven-hexagon. The reference implementation has been adapted to fit the requirements of this system. The choice of adapting it to the Onion architecture was made to allow for some consistency between the mobile application architecture and the server architecture, in hopes of lessening the required amount of knowledge new developers have to learn.

16.1 Modules

Modules are a core concept of the Nest Framework. In the context of Nest, a module is a logical grouping of code related to a specific domain [118]. A Nest module allows the developer to specify what modules a module depends on and which HTTP controllers it contains. Furthermore, they allow developers to specify what module resources are available for injection outside of the module. An injectable resource is called a provider. By default, module providers are only available within the module they are instantiated in. However, they can be made available for use in other modules by explicitly exporting them.

Code listing 16.1.1: Team module.

```
@Module({
  imports: [TypeOrmModule.forFeature([TeamOrmEntity]), UserModule],
  controllers: [TeamController],
  providers: [TeamRepository, teamServiceProvider],
  exports: [teamServiceSymbol, TeamRepository],
})
export class TeamModule {}
```

An example of a module definition is shown in Figure 16.1.1. The team module imports a user and a TypeORM module, allowing it to access any providers exported by them. Furthermore, it explicitly declares what providers and controllers the module provides. Finally, it declares a set of providers that can be used in other modules. The aforementioned usage of modules are described by the Nest team as "feature modules" [118]. Nest also provides a root module, which is responsible for specifying and instantiating the modules provided by the system.

16.2 Dependency Injection

Nest provides built-in dependency injection capabilities through the use of an IoC container. When the application is first started, Nest inspects every module definition and instantiates the providers the modules define. Classes can be marked as providers through the *Injectable* annotation. The actual instantiation of a provider is handled by Nest, but can be altered by the developer.

Code listing 16.2.1: Instantiation of farm service.

```
export const farmServiceSymbol = Symbol('farmService');
export const farmServiceProvider: Provider = {
  provide: farmServiceSymbol,
  useFactory: (
    farmRepo: IFarmRepository,
    userService: IUserService,
  ): IFarmService => {
    return new FarmService(farmRepo, userService);
  },
  inject: [FarmRepository, userServiceSymbol],
};
```

Listing 16.2.1 details how the developer is able to specify how a provider should be instantiated, what dependencies it requires and what token it should be provided as. To allow for retrieval later-on, every provider is associated with an unique token specified in the *provide* parameter. Provider instantiation is consistent throughout the system, and is only accessed through the use of symbols. The farm service is dependent on a farm-specific repository and the user service, represented through the symbol associated with it. Whenever another provider requests the farm service symbol, the *useFactory* function will run again, and return a new instance of the provider to the requester.

16.3 Identity Provider Communication

Registering a new user for the system is a two-step process. The user must first register an account with Auth0, the identity provider used throughout the system. Through the Auth0 user interface, the user is only able to register an email and a password. The system also requires users to define their first and last name, to allow for team members to identify one another. The author decided to split these

two tasks entirely; Storage of user credentials are handled by Auth0, whereas the server stores personally identifiable information. User credentials are very sensitive, and should be handled with care. By delegating the storage of sensitive information to a specialized service, like Auth0, the chance of experiencing a credentials leak is significantly reduced.

The major drawback introduced by choosing to separate the collection of user information in such a way is the synchronization between the two parts. This is solved by introducing a backend service solely dedicated to communicating with the identity provider, and orchestrating the user service to act in such a way that changes within the identity provider always carries over to the server database, and vice versa. At the time of writing, this is mostly related to deleting user accounts, although the application might grow to include functionality such as changing login credentials at a later point.

The separation of user information storage could have been entirely avoided by creating custom functionality within the identity provider. However, the author believed this to be a bad idea, as it would create a tight coupling between the server and the identity provider. By separating the registration process into two steps, changing the identity provider is an implementation change, as opposed to a complete overhaul.

16.4 HTTP Controllers

Within the context of most RESTful API's, a controller is a class handling incoming requests and returning corresponding responses. Nest allows developers to define an API controller by annotating a class with the *@Controller()* decorator. A controller can in turn expose a set of functions corresponding to REST HTTP methods. An example of this is shown in Listing 16.4.1, in which a GET endpoint for a specific team is shown.

Code listing 16.4.1: A handler for a GET HTTP method within the team HTTP controller.

```
@Get(':id')
async findOne(@Param('id', ParseIntPipe) id: number): Promise<TeamResponse> {
  const res = await this.teamsService.findOne(+id);
  return new TeamResponse(res);
}
```

The function in question is able to automatically parse and validate an ID integer, and forward the request to the team service. Once a response is received from the service, the function transforms it into the correct format and returns it to the requester. Within this application, controllers serve very specific purposes. They are entirely unaware of how any underlying processes are performed. A controller should only be concerned with authentication, DTO validation and service response transformation. Any other logic should be delegated elsewhere.

16.5 Application Services

Whereas application services in the mobile application usually centered around achieving specific use cases, the ones present on the server are more tailored towards achieving mutations within the domain it is placed in. Application services are mostly concerned with transforming DTO's into a format expected by the repository it relates to. Application service functionality related to database mutation also introduces a set of authorization checks to ensure that the requesting user is allowed to alter the affected entities through ownership or administrative rights.

16.6 Documentation

Server documentation is created through the use of TypeDoc, a library allowing one to convert TypeScript comments into HTML documentation [119]. TypeDoc documentation can be generated by running the command *npx typedoc src*. The server also provides documentation through the use of Swagger, a specific type of documentation describing the available endpoints, their parameters and their possible response types [94]. Endpoints and response types are documented through the use of Swagger annotations provided by Nest, and are shown in Listing 16.6.1.

Code listing 16.6.1: Swagger annotations for a response. Constructor omitted for brevity.

```
export class InviteResponse extends ResponseBase {
  constructor(props: InviteEntity) {
    ...
  }

  @ApiProperty({ type: () => UserResponse })
  user: UserResponse;
  @ApiProperty({ type: () => TeamResponse })
  team: TeamResponse;

  @ApiProperty({ example: true })
  pending: boolean;

  @ApiProperty({ nullable: true })
  accepted?: boolean;
}
```

In order to encapsulate package usage as much as possible, Swagger annotations are only utilized throughout the *infrastructure* layer; They are only used on HTTP Controllers, DTO's and response objects. The package also allows one to serve the automatically generated Swagger documentation in a Swagger user interface, as shown in Figures 16.1 and 16.2. By interacting with the user interface, the user is able to view extensive details about the available or required user inputs, and what every call to the server might result in. Every endpoint specifies response types and example values for different HTTP response scenarios. Finally, Swagger UI allows one to send actual requests to the server.

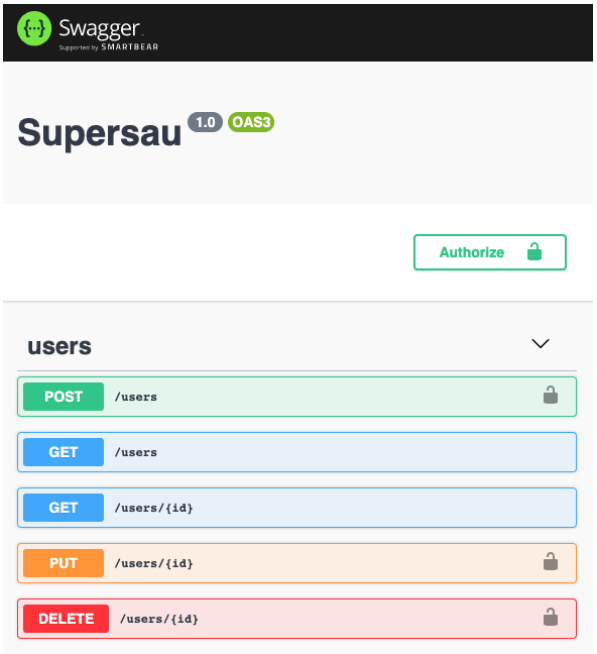

Figure 16.1: Swagger user interface.

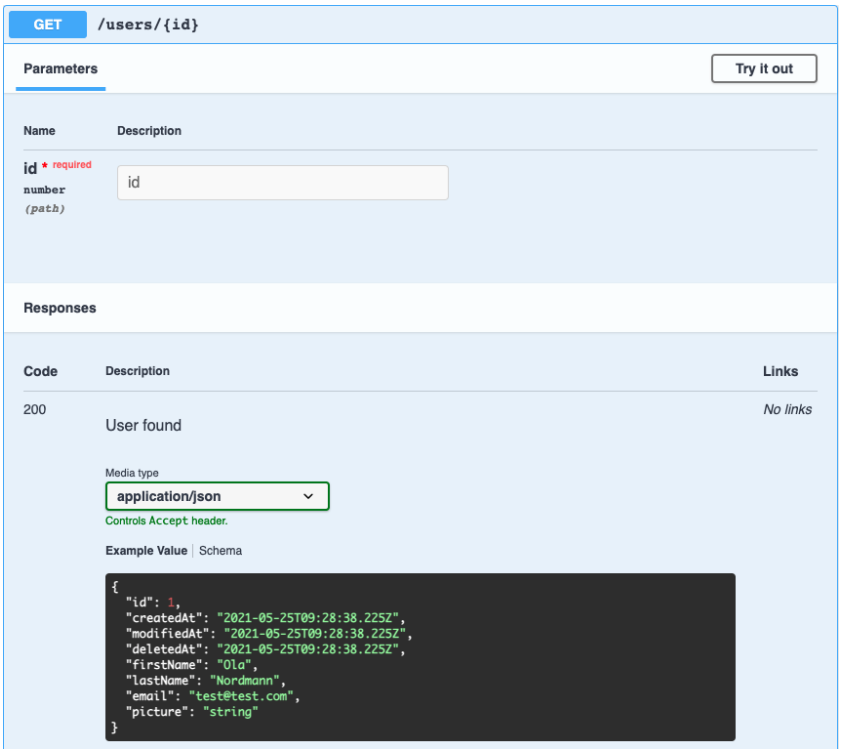

Figure 16.2: Swagger UI for getting a specific user.

16.7 Authentication and Authorization

In order to ensure that an actor interacting with the system is authorized to perform an action, the actor must first be identified and authenticated. Throughout the system, authentication is performed through the use of Auth0. Once authenticated, the actor receives a JSON Web Token (JWT), an encrypted token meant for verifying the identity of a user. The JWT can be used to interact with the server.

Code listing 16.7.1: Authentication strategy.

```
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor() {
    super({
      secretOrKeyProvider: passportJwtSecret({
        cache: true,
        rateLimit: true,
        jwksRequestsPerMinute: 5,
        jwksUri: `${process.env.AUTH0_ISSUER_URL}.well-known/jwks.json`,
      }),

      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      audience: process.env.AUTH0_AUDIENCE,
      issuer: `${process.env.AUTH0_ISSUER_URL}`,
      algorithms: ['RS256'],
    });
  }

  validate(payload: ApiUser): ApiUser {
    return payload;
  }
}
```

Server-side authentication is performed through the use of *Passport.js*, an authentication middleware for Node [114]. Nest provides functionality for easily integrating Passport.js with any Nest application. Every mutation function defined within an HTTP Controller is annotated with Passport-specific annotations. Whenever a request is received on an annotated endpoint, it is passed to an authentication strategy responsible for verifying the validity of the JWT. This strategy is shown in Listing 16.7.1. It fetches a set of keys from the identity provider (Auth0) that are used for verifying the validity of the JWT in question. Invalid JWT's will result in an error, and the request will be rejected by the server. If the JWT is valid, on the other hand, the JWT will be deserialized and passed on to the next handler in the chain of responsibility. As shown in Listing 16.7.1, the JWT verification strategy has been extensively parameterized. If one were to switch identity providers, one would only have to change the values of the utilized environment variables.

Once the validity of the JWT token has been asserted, and the request has made it to the endpoint handler, authorization is performed. The deserialized JWT contains an unique user identifier, which is also stored within the server database to simplify the authorization process. Whenever a request attempts to mutate an entity, the server checks whether the requesting user is allowed to mutate the affected entities.

16.8 ORM Entities

TypeORM allows for one to define ORM entities through the use of code, which in turn are used to generate database tables. Listing 16.8.1 shows the definition of the Farm ORM entity. Simple types, such as strings, are implicitly converted into their corresponding database types. More complex values, like enums, require some extra type specification in order for TypeORM to understand what type it should be. Due to the usage of PostgreSQL, TypeORM is able to create a specific database type for every object, ensuring that only valid values are stored. TypeORM also allows for one to define the relations an entity has to other entities. In the aforementioned example, the farm entity specifies a many-to-one relation to the User ORM entity and a one-to-many relation to the Farm Observation ORM entity.

Code listing 16.8.1: Farm ORM definition.

```
@Entity('farm')
export class FarmOrmEntity extends BaseOrmEntity {
  constructor(props: FarmOrmEntity) {
    super(props);
  }
  @Column({ length: 100 })
  name: string;

  @ManyToOne(() => UserOrmEntity, (user) => user.farms, {
    eager: true,
  })
  @JoinColumn({ name: 'owner_id' })
  owner?: UserOrmEntity;

  @Column({ name: 'owner_id' })
  ownerId?: number;

  @Column({ type: 'enum', enum: FarmColors, name: 'first_color' })
  firstColor: FarmColors;

  @Column({ type: 'enum', enum: FarmColors, name: 'second_color' })
  secondColor: FarmColors;

  @Column({ name: 'sheep_amount' })
  sheepAmount: number;

  @Column({ name: 'sheep_adult_amount' })
  sheepAdultAmount: number;

  @OneToMany(() => FarmObservationOrmEntity, (obs) => obs.farm, {
    cascade: ['soft-remove'],
  })
  farmObservations?: FarmObservationOrmEntity[];
}
```

Figure 16.3 displays all database tables the server utilizes. The structure of the tables largely resembles the ones described for the mobile application in Section 10.2.2, albeit with some slight exceptions. As such, the purpose of every table will not be discussed to the same extent.

The mobile application tables were constrained in terms of available SQLite data types, whereas the server utilizes a broader range of values. Furthermore, the server database contains a table used for keeping track of team invites. By default, a new invitation has pending set to true, and accepted set to null.

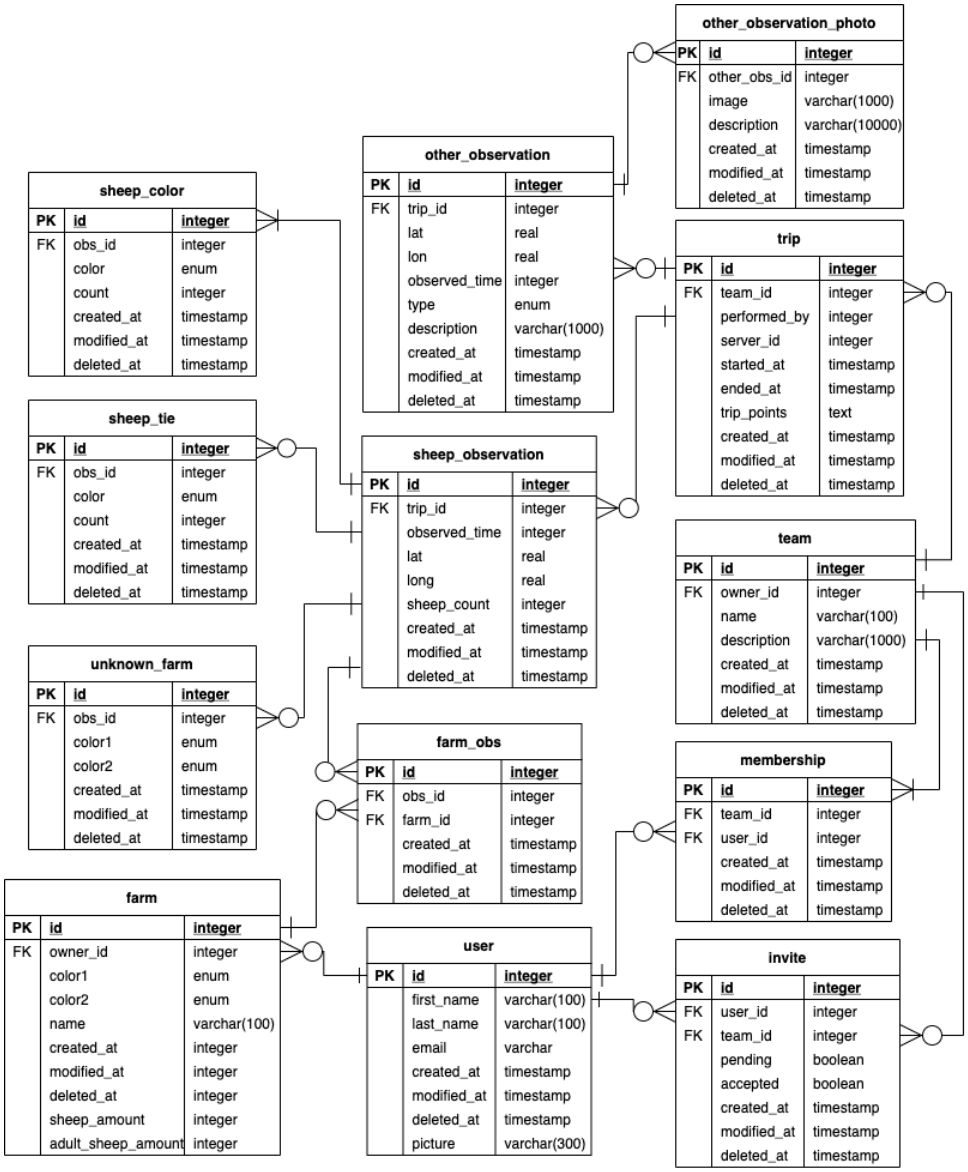

Figure 16.3: ER diagram of server.

16.9 Server Repositories

In contrast to the mobile application, the server repositories are directly responsible for persisting data to the database. This is largely caused by the usage of TypeORM; The ORM allows the developer to define database tables through the use of code. Once the tables are generated, TypeORM provides the developer with a set of functions for mutating the database. For common use cases, ORM usage abstracts away the entire database layer, allowing for a more efficient development process.

At the very core of the infrastructure layer is an abstract and generic base repository, defining functionality for persisting, fetching and deleting entities from the database. Every repository within the server extends this class. In order to extend it, a set of types must first be provided: The domain entity, the ORM entity and a set of query parameters capable of constraining entity query results. Furthermore, every repository implementation must provide an implementation of *prepareQuery*, a function that converts a set of query parameters into a format TypeORM accepts. Every entity repository implementation must provide a set of relations used for loading and deleting entities, to ensure that all required tables are included in the query results. Listing 16.9.1 provides an excerpt of the base repository, and details functionality for persisting and finding entities.

Code listing 16.9.1: Excerpt of abstract repository.

```
export abstract class BaseRepository<
  Entity extends BaseEntityProps,
  Params,
  OrmEntity
> implements IBaseRepository<Entity, Params> {
  constructor(
    protected readonly repository: Repository<OrmEntity>,
    protected readonly mapper: BaseOrmMapper<Entity, OrmEntity>,
  ) {}
  protected abstract relations: string[] = [];
  protected abstract prepareQuery(
    params: QueryParams<Params>,
  ): WhereCondition<OrmEntity>;

  async save(entity: Entity): Promise<Entity> {
    const ormEntity = this.mapper.toOrmEntity(entity);
    const result = await this.repository.save(ormEntity);
    return this.mapper.toDomainEntity(result);
  }

  async find(params: QueryParams<Params> = {},withDeleted = false,)
  : Promise<Entity[]> {
    const where = this.prepareQuery(params);
    const found = await this.repository.find({
      withDeleted: withDeleted,
      relations: this.relations,
      where,
    });
    return found.map((e) => this.mapper.toDomainEntity(e));
  }
}
```

In trivial cases, the abstract repository enables one to create a repository with only a few lines of code. For more complex use cases, a specific repository can either choose to override the default implementations or define new functionality.

Another core component of the abstract repository implementation is the entity mapper. Domain and ORM representations do not always align. Thus, one must be able to transform domain entities to ORM entities and vice versa. The abstract repository makes use of an abstract entity mapper, whereas every repository implementation provides an actual implementation of an entity-specific mapper. In order to reduce boilerplate code, the abstract entity mapper maps properties shared by all entities, namely the ID and the created, modified and deleted dates.

Entities within the server are never deleted; Instead, they are omitted from query results, a functionality provided by TypeORM. Due to the need for synchronizing the state of a mobile application with the state of the server, the server must be able to keep track of what entities have been deleted since a user last synchronized.

16.10 Responding to Requests

The user should not always receive a directly serialized version of the actual entity in response to a request. In order to decouple the actual implementation of an entity and the values returned to the user, entities are transformed to dedicated response objects. Every response object specifies a set of properties a successful response should return. By utilizing such an approach, the developer ensures that responses only contains data intended for the end user. Alternatively, one could choose to omit specific properties of an entity. The author believed this approach to be more error-prone; Adding new properties could suddenly impact the response.

16.11 DTO Validation

As mentioned in Section 15.1.1, user input is handled through the use of Data Transfer Objects. Every DTO contains data validation rules to ensure the correctness of any data processed by the server.

Code listing 16.11.1: Definition of validation logic for team creation.

```
export class CreateTeamDto {
  @ApiProperty()
  @IsString()
  @Length(2, 500)
  name: string;

  @ApiProperty()
  @IsOptional()
  @IsString()
  @Length(0, 500)
  description: string;
}
```

The server utilizes *class_validator*, a JavaScript library allowing for one to validate the contents of an object. When combined with NestJS, DTO validation can be performed automatically whenever a request is received. An example of DTO validation is shown in Listing 16.11.1. If the user input in a DTO breaks a validation rule, the request is aborted and a descriptive error message is returned to the requester. An example of this is shown in Figure 16.4.


```
{
  "statusCode": 400,
  "message": [
    "name must be longer than or equal to 2 characters"
  ],
  "error": "Bad Request"
}
```


Figure 16.4: Error response returned when DTO validation fails.

16.12 Synchronization

In order to enable offline capabilities for the mobile application, the server must be able to provide a user with all changes in database state relating to the user since the last time a synchronization occurred.

All entities are synchronized at the same time; The application does not allow for synchronizing one entity at a time. This approach was chosen due to the nature of the relationships between entities. A change in one entity often results in changes in another. If a user has joined a team since last synchronizing, the synchronization process must provide the new memberships, users and team trips for that team. Database synchronization is already a complicated process. It is possible to optimize it further than it currently is, but the author believes the performance-to-complexity trade-off to not be worth it at the current time.

As discussed in Section 10.5, the synchronization endpoint expects to receive a set of dates representing the time the user last synchronized with the server. The dates are separated by entity to allow for the implementation to fetch entities in separate queries if required, ensuring the validity of the results with regards to database changes happening throughout the synchronization process. Furthermore, the synchronization functionality also allows for the user to pass in a list of locally downloaded farms that should be synchronized. The synchronization endpoint requires authentication, allowing the server to extract the authentication ID of the user performing the request. This ID is utilized to fetch the teams the user is a member of, and acts as the foundation for the entire synchronization process. Based on the results of the team retrieval process, the synchronization service

fetches, filters and partitions memberships, users, farms and trips. Each entity is separated into three distinct categories, namely created, modified or deleted. Created and modified entities are returned in their entirety; Deleted entities are only returned as identifiers to allow for a lower amount of transferred data.

16.13 Image Storage

"Other" observations should allow for the sheep observer to attach images of their findings. As such, the system must support the storing and sharing of images. The author decided to offload the task to AWS S3, an AWS service for storing objects [120]. The server receives images whenever a user decides to upload a trip. Once received, images are processed and uploaded to S3 one by one. Finally, the image data is replaced with the URL of the stored S3 object before being persisted to the database. This approach brings several benefits over simply storing the images in a database. The size of the database would increase drastically if one were to store images within it. This could be somewhat mitigated through image compression, but not nearly enough to warrant the cost. The performance penalties usually involved with storing files in databases are growing smaller, but can still be felt once a critical mass is reached. Furthermore, database storage tends to be more costly than file system storage in cloud services. Finally, and possibly most importantly, offloading the process of image retrieval to another service would increase the throughput of server requests. By not having to perform costly read operations with regards to images, the server will have more resources for handling simpler requests. This implementation introduces the drawback of having to synchronize the state of the database with the state of the remote file storage system. However, the author believes this to be a negligible drawback compared to the benefits brought by the approach.

Chapter 17

Testing the Server

The process of testing the server is somewhat intertwined with the mobile application testing process described in Chapter 11; System testing either directly or indirectly makes use of the server. As it is already covered in Chapter 11, it will not be discussed here. This chapter will describe the process of checking the logical soundness of the server source code.

Server tests are performed through the use of *jest* [121], a testing framework for JavaScript. It is used alongside *ts-jest* [122], which allows for TypeScript code to be tested through *jest*. Dependency stubs are created through the use of *sinon* [123], a package for creating fake, mocked or partial implementations of JavaScript and TypeScript classes. In order for a test to be picked up by *jest*, it has to be appended with a *.spec* ending. This naming is consistent through the server; A test suite testing *user.repository.ts* would be named *user.repository.spec.ts*. Tests are stored in a separate folder mirroring the actual structure of the server source code.

In order to ensure the correctness of the server persistence implementation, one must test the integration between the server repositories, TypeORM and the underlying database. Persistence is tested against an actual PostgreSQL database solely dedicated to testing. This is largely related to the use of TypeORM; If one were to mock out the entire TypeORM library, there would be no way to actually ensure that entities are persisted, mutated and retrieved correctly. The mobile application utilizes SQLite, a lightweight database RDBMS with the capability of running entirely in-memory. Creating and destroying in-memory databases is faster than creating a traditional database. However, PostgreSQL cannot be ran in-memory. As such, persistence-related tests are ran against a fully-fledged PostgreSQL instance.

The persistence tests expects a PostgreSQL test instance to be available locally at port 5443. This project has handled database instances through the use of Docker Compose, a tool allowing for the developer to provision several resources at a time. The test database has, for the most part, been provisioned together with the rest of the server development environment. However, it can also be provisioned as a standalone instance with the command *docker-compose up test_db*.

An example of a persistence test is shown in Listing 17.0.1. The first part of the listing shows how the database is handled in-between tests. Before any test within a test suite is ran, a new connection to the database is created. Before every test is ran, the database is dropped and rebuilt to ensure that tests do not affect one another. After all the tests in the test suite has completed, the database connection is closed. The latter part of Listing 17.0.1 displays a typical persistence test. It simply creates a new user entity, persists it to the database and asserts that it has been given an identifier.

Code listing 17.0.1: Testing user persistence.

```
describe('The User Repository', () => {
  let conn: Connection;
  let userRepo: IUserRepository;
  beforeAll(async () => {
    await connection.create();
    conn = getConnection();
  });

  afterAll(async () => {
    await connection.close();
  });

  beforeEach(async () => {
    await connection.clear();
    userRepo = new UserRepository(conn.getRepository(UserOrmEntity));
  });

  it('should be able to insert unique user', async () => {
    const props: UserProps = {
      authId: 'auth0||asdasdsa',
      email: 'test@test.com',
      firstName: 'Jonas',
      lastName: 'testing',
      picture: 'https://test.com/this.png',
    };
    const user = new UserEntity(props);
    const res = await userRepo.save(user);
    expect(res.id).not.toBeNull();
  });
});
```

HTTP controllers are tested through the use of *supertest* [124], a framework allowing for one to easily test HTTP endpoints. A controller test suite contains tests for every responsibility a controller can have: It asserts if the controller is capable of parsing and validating DTO's, transforming responses and rejecting

invalid requests. Listing 17.0.2 describes the process of instantiating the Team HTTP controller inside of a test module. Controller dependencies are replaced with stubbed versions, allowing for one to specify the outcome of every service response. Authentication is also mocked. Whether a request is authenticated or not is controlled through the use of two mock functions, each changing the outcome of the authentication process.

Code listing 17.0.2: Setting up the Team HTTP Controller test. Validation Pipe omitted for brevity.

```
describe('Team Controller', () => {
  let app: INestApplication;
  const teamService = sinon.createStubInstance(TeamService);
  let returnMock = jest.fn();
  let throwMock = jest.fn();
  beforeEach(async () => {
    returnMock = jest.fn();
    throwMock = jest.fn();
    const testAppModule = await Test.createTestingModule({
      controllers: [TeamController],
      providers: [
        {
          provide: teamServiceSymbol,
          useValue: teamService,
        },
      ],
    })
      .overrideGuard(AuthGuard('jwt'))
      .useValue({
        canActivate: (context: ExecutionContext) => {
          if (throwMock()) {
            throw new UnauthorizedException();
          }
          const req = context.switchToHttp().getRequest();
          req.user = validUser;
          return returnMock();
        },
      })
      .compile();

    app = testAppModule.createNestApplication();
    await app.init();
  });
});
```

Listing 17.0.3 provides two examples of controller tests. The first test checks whether the deletion endpoint requires an user to be authenticated. If the endpoint is annotated correctly, the authentication mock solution will result in the endpoint returning a 401 error, indicating that the request was unauthorized. The second test checks whether the controller will call the underlying team service with the correct parameters when provided with valid inputs. It specifies a predefined team that the team service mock should return whenever the *create* method is called with a set of specific values.

Code listing 17.0.3: Controller test examples.

```
describe('Team Controller', () => {
  beforeEach(async () => {
    ...
  });

  describe('delete', () => {
    it('requires authentication', async () => {
      throwMock.mockReturnValueOnce(true);
      await request(app.getHttpServer()).delete('/teams/1').expect(401);
      expect(throwMock).toHaveBeenCalledTimes(1);
    });
  });

  describe('create', () => {
    it('can successfully call service', async () => {
      throwMock.mockReturnValueOnce(false);
      returnMock.mockReturnValueOnce(true);
      const team = createTeam(1);
      const res = JSON.stringify(new TeamResponse(team));
      const dto = {
        name: 'My Team',
      };

      const actualDto: CreateTeamHttpDto = {
        name: 'My Team',
      };

      teamService.create
        .withArgs(sinon.match(actualDto), validUser.sub)
        .resolves(team);
      await request(app.getHttpServer()).post('/teams').send(dto).expect(res);
    });
  });
});
```

Chapter 18

Server Deployment

In order to allow for one to easily modify or extend the server, the process of deploying a new and production-ready version of the server has to be as easy as possible. This project realizes this through the use of continuous deployments whenever a new revision of the source code is pushed to the master branch of the source code repository. In short, AWS fetches new commits, builds a new server artifact and runs it. The infrastructure making this possible is described later-on in Section 21.2. This functionality is realized through the use of a set of files written with the sole purpose of simplifying deployment to the AWS infrastructure, and will be described in the following paragraphs.

The source code repository has a so-called web-hook: A specific functionality that is performed every time an action occurs. In the case of the server repository, this web hook is responsible for notifying AWS whenever a new revision of the source code is pushed to GitHub. AWS will in turn fetch the new source code revision and build a new server artifact from it. The artifact build process is controlled through the use of *buildspec.yml*, a file specifying a set of steps that will result in a new server artifact.

In large, the artifact build process specified in *buildspec.yml* consists of two phases. The first phase solely consists of downloading every project dependency, and is required in order for the artifact to be built. In the second phase, the actual server artifact is built. This is achieved through a command specified in *package.json*, a file home to a set of scripts the server supports. This script is later copied over to the artifact to allow for it to understand how to start itself later-on. Once the server artifact has been created, the process goes on to remove the dependencies that were only required for the artifact build process. Runtime dependencies are copied over into the build artifact.

The build process also consists of creating a file filled with environment variables

not known at the time of deployment. These variables mostly revolve around non-static or secret values fetched from AWS, such as database passwords, Auth0 credentials and the port the server should receive requests from. Sensitive variables are fetched from a secure storage solution provided by AWS, as discussed in Section 22.2.2 later-on. Non-sensitive values are simply injected into the build environment.

A set of AWS-specific files are also copied over to the build artifact, namely *appspec.yml* and corresponding installation scripts. The file simply serves as a place in which one can define a set of scripts to be ran at different parts of the server instantiation process. Once the build process is complete, the artifact will be stored within AWS, which in turn will notify the computational devices the server runs on. These devices will fetch the newly created server artifacts, and perform the set of scripts specified in *appspec.yml*. As with the build process, the initalization process consists of two phases. The first phase simply downloads and installs Node so that the server can run. Furthermore, it runs a set of commands that allows Node processes to listen to privileged ports without being ran as an elevated process. The second phase simply kills all running node instances before finally starting a new server instance.

18.1 Running Locally

The server can also be ran locally during development. This has been achieved through the use of Docker compose, which creates three Docker containers: A PostgreSQL database used for testing, a PostgreSQL database used for running the application locally and a container for running the actual server. The entire stack can be started by running *docker-compose up* within the server directory.

The server expects a set of environment variables to be present in order to run locally. The non-test database utilizes *database.env*, which defines a database username, a database password and a database name. Server-related environment variables are defined in a file named *.env*. It expects an environment variable specifying what port the server should run on locally, and the name of the AWS S3 bucket to store images to. Furthermore, it expects a set of environment variables relating to Auth0, so that incoming requests can be authenticated. In order to enable identity provider communications, one must provide the URL of the token issuer, the audience requesting access tokens and the client credentials the server uses to communicate with Auth0. Finally, the server requires the same environment variables as the database does, so that it can correctly connect and perform queries.

Chapter 19

Server Discussion

The server was developed at a more rapid than the mobile application, a process largely expedited through the use of powerful packages and familiar technology. This chapter will provide a retrospect of the choices the author felt to be the most consequential for the final product.

19.1 ORM Usage

Choosing a persistence solution is a highly contested topic within the Node community, with every developer favoring different approaches. In the grand scheme of things, the community is split on the usage of ORM's, query builders and pure SQL, ranked descending in terms of their level of abstraction. The author decided to utilize an ORM due to the increase in developer productivity it could provide. Furthermore, using an ORM would allow for one to switch out the underlying database without having to make any changes within the code base, increasing the flexibility of the system. The author found the ORM usage to be very beneficial when developing the server, and can largely be attributed to two factors: Soft deletion and entity insertion.

Due to the synchronization requirement of the system, the server must be able to keep track of deleted entities. An entity cannot simply be deleted, as it would make the state of every mobile application storing the entity invalid. As such, the server must be able to "soft delete" entities, a process in which the entity is persisted in the database, but not accessible through ORM queries. This functionality can be implemented without the usage of ORM's, but there are a lot of caveats one will have to keep in mind. This was, in fact, a large reason as to why the author decided to utilize TypeORM, as it had built-in support for cascading soft deletes.

The benefit of ORM insertion is best described through the process of inserting a trip. Every trip can have a number of sheep observations and "other" observations.

A sheep observation can have several sheep ties, sheep colors, farm observations and unknown farm observations. Furthermore, an "other" observation can contain several observation photos. One can image inserting the data related to a trip to be a complicated process including a lot of SQL statements. Instead of inserting every entity sequentially in different statements, an ORM allows the developer to insert the entirety of the trip entity with all related data in one single function, without having to write any insertion logic.

ORM usage does have some caveats. Utilizing an ORM usually involves a performance penalty, as generated queries are not optimized for the table it targets. One could make a case for taking a more bare-bones approach to entity persistence, by for instance utilizing an approach more akin to prepared SQL statements. A similar case could be made for query builders, which acts as a middle ground between prepared SQL statements and ORM's. Constraints introduced by ORM usage are not necessarily transparent when choosing the approach. If the project at any time encounters a constraint large enough to invalidate the choice of an ORM, the technical debt attributed to switching approaches could be large. This is somewhat alleviated by the choice of encapsulating ORM usage, allowing for portions of the server to be modified independently.

Utilizing an ORM can greatly reduce development time as long as the intended functionality is common and supported by the ORM. For more complex and uncommon functionality, raw SQL queries might be the better choice. The aforementioned drawbacks have not affected the server at the time of writing. However, one should be aware of their existence, and always look for better options.

19.2 Choosing Nest

The overall composition of package choices made throughout the server is largely associated with the choice of utilizing Nest as the web application framework. In large, the choice of Nest has proven to be sound, as it has been able to fulfill every server requirement through custom functionality or integrations with other packages. Nest was simply chosen due to it providing integrations with a set of functionalities the server would require at some point. The built-in support for Swagger, request validation, authentication and dependency injection provided a drastic reduction in development time over more bare-bones alternatives.

Arguments could be made for choosing other frameworks over Nest. The benefit of choosing Nest is that it primarily builds upon popular packages in the Node ecosystem. As long as Nest usage is kept somewhat concentrated, one can migrate away from it at a later point. Nest migration can be performed incrementally, by refactoring integrations to instead rely directly on the package Nest utilizes. For the most part, Nest usage has been contained within the infrastructure layer. The application core would remain largely unaffected if one were to switch frameworks.

Part IV

Cloud Infrastructure

Chapter 20

An Introduction to the Cloud

In the modern landscape of applications and systems, usage of "the cloud" has become nearly ubiquitous. As systems and requirements grew to be more complex, the traditional approach of simply hosting it yourself became a major roadblock to many. Can your infrastructure withstand an attack? How are the different attack surfaces protected? Will the server be able to scale in response to an increase or decrease in traffic? How will it handle failure? In the recent decade, the most common solution has been to choose a cloud provider: A vendor offering servers and flexible solutions allowing developers to easily host anything. The larger cloud providers have a plethora of different solutions, ranging in size and complexity. As a developer, you are able to choose between a fully preconfigured solution only requiring the push of a button to host an application, all the way down to a fully configurable computational instance.

This part will detail the cloud infrastructure this thesis utilizes. It will first provide a quick primer on cloud technology before describing the infrastructure this system provisions. Following this, it will detail how the infrastructure can be deployed. The part is concluded with a discussion surround the choices made when creating the cloud infrastructure.

20.1 The Cloud: A Primer

The cloud can, in simple terms, be described as a set of servers connected through the internet, allowing world-wide access to whatever the servers in question provides. In the early days of the internet, many companies hosted their systems on their own hardware. This eventually lead to data centers, a set of servers owned and maintained by a company. The server operating systems allowed for a technology called virtualization, a concept allowing several systems to share the underlying machinery. Through the use of virtualization, data centers could rent out a slice of

a server to a company. The data center could charge a premium for providing and maintaining the servers, essentially removing the need for the renting companies to maintain their own server infrastructure. In the rest of this thesis, this practice will be referred to as Infrastructure as a Service (IaaS).

As the internet grew in size and complexity, as did the requirements of the systems running on these servers. In the past two decades, the practice of IaaS has been expanded upon and simplified for more specific use cases. For instance, Platform as a Service (PaaS) provides a deployment-ready development environment, allowing customers to not have to worry about the configuration and maintenance of the underlying operating system. An even simpler "as a service" solution is the category of Software as a Service (SaaS), in which a company provides a production-ready software solution the customer can configure to their specific needs. In the context of this project, a viable SaaS solution could for instance be a storage solution.

Companies providing IaaS, PaaS or SaaS have come to be known as cloud providers. Within this category, the three biggest companies are Amazon (Amazon Web Services), Microsoft (Azure) and Google (Google Cloud Platform). Each provider boasts a large amount of services at comparable prices. Thus, choosing one over the other can be a difficult affair. In all but the most specific cases, each cloud provider has a service fulfilling any requirement. If the system is Windows-based, one could make a sound argument for choosing Microsoft Azure, although both AWS and GCP also provides Windows-based resources.

Having previous experience with AWS, the platform was deemed to be the most appealing to use as the learning curve would be lower. However, the author decided to research all three providers before making a final decision. After all, cloud provider services are similar enough to allow for a transferal of knowledge. The research process consisted of mapping out the required infrastructure resources and services the project required, followed by reading about the alternatives each cloud provider offered. No cloud provider stood out in particular, as all requirements were covered by each of them. If one were to simply choose a cloud provider to have a place to host a simple system, Microsoft Azure provides what the author considers to be the best solution due to a simple and clear user interface. However, this project will provision all infrastructure through the use of code, using a process known as Infrastructure as Code (IaC). As such, the simple user interface Azure provided added no benefit to the project.

The choice eventually boiled down to price. Each provider offers a generous free tier, allowing new users to utilize certain services and resources in a limited manner. If you utilize the platform in a way not covered under the free tier, you will be charged. However, Microsoft Azure and AWS provides student tiers, allowing one to have more freedom to experiment. Microsoft Azure provides students with free tier usage and $100 in credits, whereas AWS provides $30 of credits. Any action outside of the free tier would detract from the credits. Once the credits are

depleted, the account is closed. The author considered these tiers to be the best option, as they would allow for experimentation without unforeseen monetary ramifications. After testing both student tiers, the author found Microsoft Azure student accounts to not provide sufficient permissions to provision the required infrastructure for the system, leaving AWS to be the most viable option. As such, the author chose to utilize AWS.

Chapter 21

System Infrastructure

This section describes the infrastructure that is provisioned for both AWS and Auth0. Nearly all infrastructure utilized by the system is provisioned through AWS. The entirety of the server lives within a Virtual Private Cloud (VPC), a virtual network isolating its resources from anything outside of the VPC unless explicitly stated otherwise. In this case, some exceptions are made to allow for everything to work as expected.

21.1 Auto-Scaling Group

The Application Programming Interface (API) is created using Elastic Compute Cloud (EC2), a compute service AWS bases most of their other compute services on. EC2 allows a user to provision a resizable virtual machine at a desired size. This flexibility allows developers to scale up the computational capacity of their instance when a bottleneck is encountered.

AWS provides a plethora of compute services, where many of them allows a developer to create clusters of compute instances with the capability of scaling up or down based on demand. This application uses an auto-scaling group to achieve this, as shown in Figure 21.1. Amazon defines an Auto-scaling group as "a collection of EC2 instances that are treated as a logical grouping for the purposes of automatic scaling and management" [125]. The developer can choose a set of constraints that, when met, can increase or decrease the amount of EC2 instances within the group. Furthermore, by using an auto-scaling group, AWS allows the developer to specify an API endpoint for verifying the health of an instance. If the endpoint does not respond as expected, the auto-scaling group considers the instance to be unhealthy, and replaces it with a new instance.

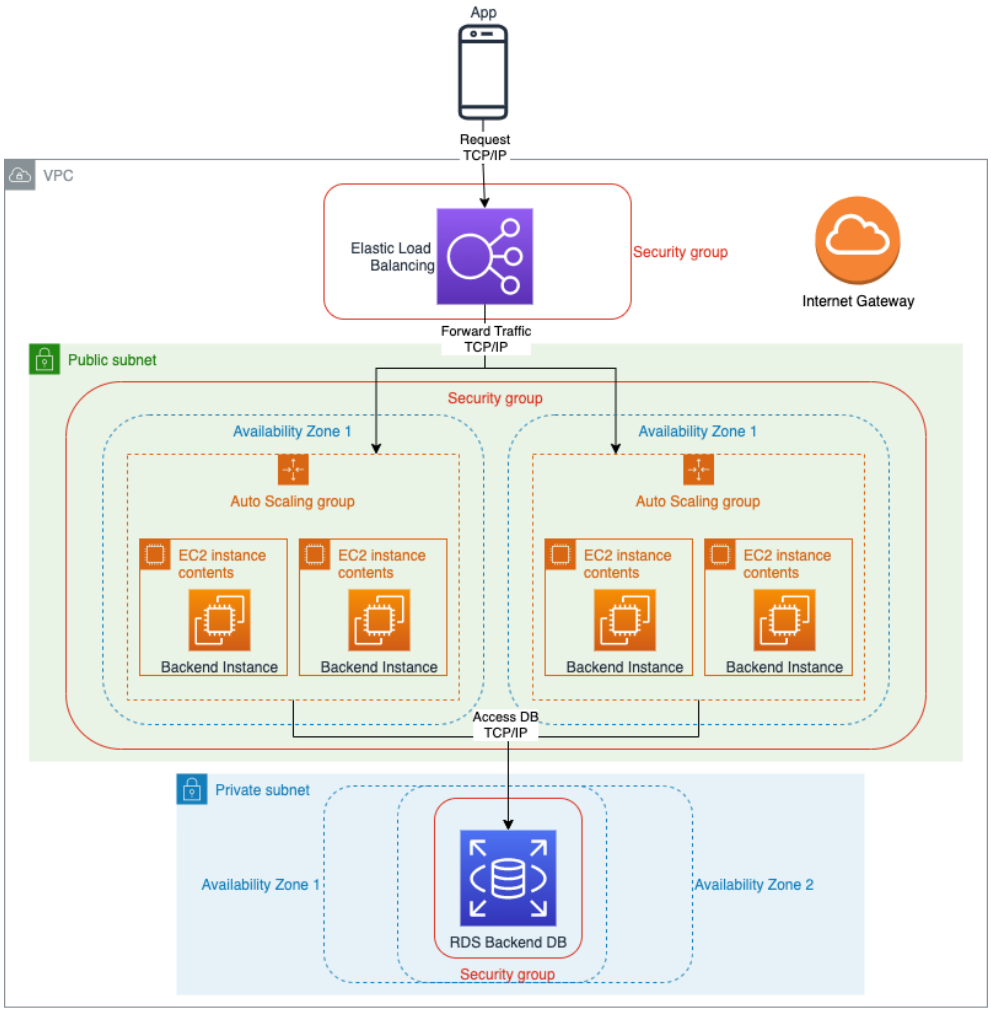

Figure 21.1: Deployment diagram of the server infrastructure.

Being able to provision or decrease computational power on the go is very beneficial. If the system is under heavy load, a new instance can be created to allow new requests to still be met at full speed. The opposite also applies; If the system is under low load, superfluous instances can be terminated to save costs.

AWS categorizes their data centers in geographical regions, which in turn are divided in different availability zones. The auto-scaling group spans several availability zones to ensure availability even if one availability zone encounters a catastrophic failure. By covering several availability zones, the application is guaranteed the highest possible availability.

In order to be able to access the instances, one has to be able to locate them and delegate traffic appropriately. AWS provides this functionality through the use of

their Elastic Load Balancer, a vital piece of infrastructure automatically distributing traffic to the aforementioned compute instances. By connecting the load balancer to an auto-scaling group, compute instances are automatically registered by the load balancer. Furthermore, the auto-scaling group can utilize metrics provided by the load balancer to scale the instance count of the auto-scaling group up or down. The load balancer is also able to pick up on failed instances, and will stop routing traffic to them. In addition to the previously mentioned benefits, the load balancer is able to scale itself to match the amount of incoming requests.

AWS provides several load balancers one can consider to be relevant to this project, namely the application, network and classic load balancers. The classic load balancer is considered legacy, and should not be used unless the system has other legacy infrastructure not supported by other load balancers. As for the network and application load balancers, the choice is more nuanced. The network load balancer operates on the transport layer of the OSI model, and is not able to make sense of any incoming requests. It simply receives the request and forwards it to an instance in the target group it points to. The Application Load Balancer, on the other hand, operates on the application layer, and is therefore able to understand the entirety of a request. This enables the developer to add additional functionality to the load balancer, such as authentication. The additional layers the application load balancer has to handle does come with a slight penalty in terms of total throughout, but it could be considered a negligible difference in the use cases of this system. In cases where throughput is high, such as video streaming or messaging, a network load balancer could provide a slight edge, but the author believes the extensibility provided by the application load balancer made it the better choice for this system. Choosing it over the network load balancer would allow for one to add authentication and authorization directly into the load balancer at a later stage.

Each availability zone covers a public subnet containing the compute instances created by the auto-scaling group. By being in a public subnet, the compute instances are able to access the internet at large, with incoming and outgoing traffic being allowed. Private subnets, on the other hand, disallows all traffic outside of the VPC. The server utilizes an external identity provider, and is therefore reliant on being able to perform network requests to it. Furthermore, the compute instances must be able to download necessary software when initially launched. As such, the compute instances will require internet access. One should be careful about granting access directly to key components of the infrastructure, as it increases the potential attack surface. AWS offers several solutions to such problems.

A potential solution is to utilize a NAT gateway, an AWS service providing instances within a private subnet access to the internet without allowing the internet to access them. This would allow the compute instances to be contained within a private subnet, whose only option of accessing the outside internet would be through the NAT gateway. However, NAT gateways could not be utilized by student accounts throughout the development period, and is therefore not utilized.

This application utilizes a public subnet with access restrictions. All outbound traffic is allowed, and routed through the internet gateway, whereas each compute instance only allows incoming HTTP traffic from the load balancer. With this approach, the compute instances are just as restricted as they would be with the NAT gateway approach, only with fewer resources. The trade-off is a loss in security configuration, but the configuration provided by the current solution has proven to be sufficient.

The final piece of the server infrastructure is the database. It is provided by the AWS Relational Database Service (RDS). As with everything else in the application, it is hosted in several availability zones, and can scale according to the required capacity. It is stored in a private subnet, and can only be accessed by instances within the server security group.

21.2 Continuous Deployment

In order to make the deployment of a new version of the server as quick and easy as possible, the infrastructure allows for automatic deployment whenever a new change is pushed to the master branch of the source code repository. The entire process is described in Figure 21.2. In the case of the server, source code is stored on GitHub. GitHub provides web-hooks that can be configured to fire whenever a specific condition is met. The server repository is configured to fire whenever a commit is pushed to the master branch of the server repository.

AWS provides a set of services that allows one to control the entirety of the CI/CD process. The most central service within this category is CodePipeline, which offers capabilities to enable the entirety of a CI/CD process, all the way from hosting a repository to deploying it to production.

CodePipeline acts as an intermediary between more specific services, namely CodeCommit, CodeBuild and CodeDeploy. CodeCommit is a Git implementation hosted by AWS, and is not used within this application. The author chose to store all source code on GitHub, mostly due to habit and familiarity.

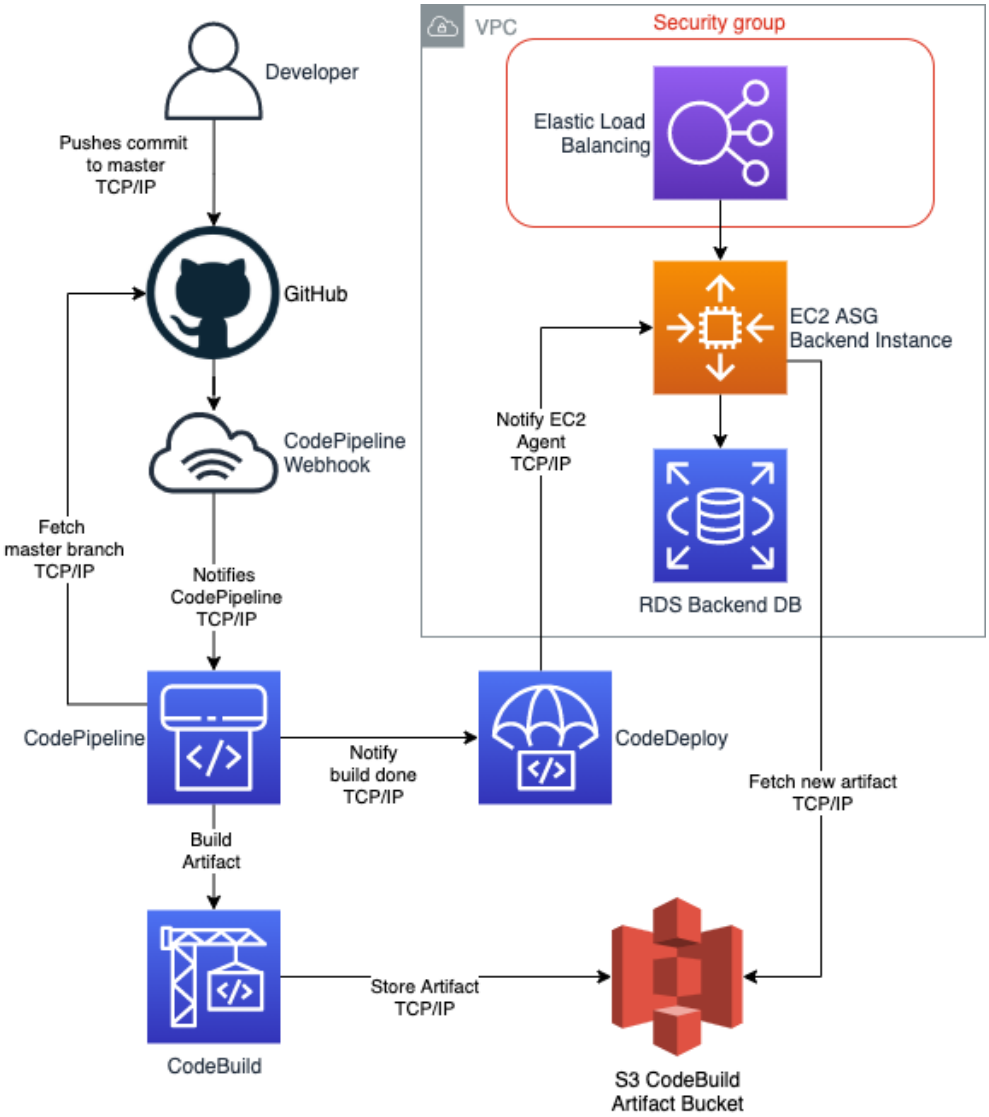

Figure 21.2: Deployment diagram of the continuous deployment process. Server infrastructure is simplified.

As previously stated, a GitHub web-hook fires an event whenever a new change is pushed to the master branch. CodeBuild, an AWS service, listens for these events, and responds by fetching the newest version of the source code. It then proceeds to use a build specification file stored in the source code to create an artifact. In the case of this project, the build process consists of installing all of the required node modules, transpiling the TypeScript files to JavaScript files and moving them to a distribution folder. Furthermore, the build process fetches sensitive information from the AWS parameter store and stores them in an environment file.

GitHub also provides excellent tooling for creating build artifacts through the use of GitHub Actions. As previously stated, the build process requires one to download and bundle all node modules required for the server to run. The author believed that it would be inefficient to do this using GitHub, and then transfer all of the node modules over the internet to AWS. Instead, the entire process could be handled within the AWS ecosystem, which would result in having to transfer less data from GitHub to AWS. One could argue that using CodeBuild ties one up too tightly with the AWS ecosystem, but the author believed this to be void. If required, the process of switching from CodeBuild to GitHub actions would be trivial.

Once CodeBuild completes the artifact, it is encrypted and stored in a private S3 bucket. CodePipeline then notifies CodeDeploy, which in turn notifies the auto-scaling group. The auto-scaling group fetches the new artifact from the S3 bucket, stops any running processes, and starts new processes with the newest source code revision.

21.3 Auth0 Infrastructure

Protecting resources from ill-intended actors is of the utmost importance in most systems. This project is no exception. The infrastructure of the provisioned Auth0 resources is much less complicated than the one provided for AWS, mostly due to the fact that Auth0 implicitly provisions most resources automatically upon creation of others. In fact, the project only explicitly provisions four resources: two clients, an Auth0 API and a client grant.

In order to allow the mobile application to communicate with the rest of the system, Auth0 requires one to define a client that allows for one to log in through the use of the native authentication flow, as defined by the IETF in RFC 8252 [65].

The server requires a different type of client in order to communicate with Auth0, namely a machine-to-machine client. This client allows two machines to communicate securely through the use of shared credentials, which both parties are able to store securely.

Finally, a custom Auth0 API is provisioned to allow the server to access Auth0. Limiting what Auth0 API's are allowed to call are specified through the use of the aforementioned client grant. In the case of this system, the server is only allowed to create, update, delete and retrieve users.

Chapter 22

Infrastructure Deployment

With the rise of cloud providers, provisioning and managing an infrastructure gradually became easier. Most cloud providers allows the user to utilize an user interface to provision resources. This allows for a simple creation process, albeit not a reusable one. If one were to clone an infrastructure to allow for different development environments, for instance, the user would have to simply remember how the original infrastructure was provisioned and copy the steps. This can oftentimes become difficult for larger infrastructures, and introduce unnecessary complexity and confusion. Thus, the principle of Infrastructure as Code (IaC) was born.

Instead of provisioning resources through user interfaces or command lines, IaC allows the user to provision resources using code. Leveraging this principle provides a plethora of benefits. For instance, one can gain a much greater overview of an infrastructure by simply reading resource definitions, as opposed to traversing user interfaces and creating a mental model. Furthermore, code usually allows for some level of variability. To expand on a previous example, creating a new environment could for instance be done by changing a variable in the code. Finally, modern IaC tools removes the need for developers to get familiarized with new user interfaces and consoles. Although the underlying resources change, the concepts of IaC remains the same.

AWS supports several IaC tools, with the first being AWS CloudFormation [126]. This service allows for users to define infrastructure using either JSON or YAML. The user creates a CloudFormation template describing the properties a certain AWS service should have. AWS allows the user to provision the requested infrastructure through the use of either the command line, their API or their user interface.

Amazon released an alternative to CloudFormation in 2019, the Cloud Development Kit (CDK) [127]. The CDK allows for developers to define their user infrastructure

using several popular programming languages, such as Python, TypeScript and Java. This provides developers with familiar programming language tooling; Type checking, code suggestions and automatic linting both simplifies and expedites the development process.

Another popular IaC tool is Terraform [128]. Unlike the previously mentioned tools, Terraform is a third-party solution, and is entirely cloud-agnostic. It achieves this by offering providers: plugins capable of creating resources on a specific platform. Each plugin provides a list of supported resources, examples of how a specific resource can be created, and a list of available parameters. Terraform allows developers to define infrastructure through a domain-specific language specifically created for Terraform, namely the HashiCorp Configuration Language.

Choosing the right IaC tool would allow for a quicker and easier development experience. The author extensively researched each tool before making a final choice. CDK appeared to be a good choice, mainly due to being able to utilize programming language tooling to expedite the development. Furthermore, by choosing to define the infrastructure in TypeScript, the infrastructure and the server would be written in the same language. However, the infrastructure needs to provision resources from outside of the AWS ecosystem. Splitting the infrastructure in two could be an alternative, where AWS resources could be defined in CDK or CloudFormation, and everything else could be defined using Terraform. This would, however, introduce unnecessary complexity and confusion, and would detract from the added benefit of being able to gain an overview of the infrastructure through reading code. This eventually led to choosing Terraform.

The author also considered two additional options: The Terraform CDK [129] and Pulumi [130]. Both options aim to solve the same problem as the AWS CDK: Define infrastructure through programming languages. However, both options are still new and unproven. Choosing a battle-tested solution ensures the rigidity of the infrastructure and lowers the chance of encountering unexpected surprises. As such, Terraform with the HashiCorp Configuration Language remained the preferred choice.

22.1 Terraform Primer

Terraform is an orchestration tool enabling the user to provision infrastructure through a domain-specific language. Under the hood, Terraform is able to make sense of the aforementioned language, and figure out what actions needs to be performed in order to reach the desired state. The AWS provider achieves this by planning a set of calls one must make to the AWS API. An example of a resource definition can be found in Listing 22.1.1, and describes the creation of an S3 storage bucket.

Code listing 22.1.1: Provisioning an S3 bucket through Terraform

```
resource "aws_s3_bucket" "codepipeline_bucket" {
  bucket = "${var.environment}-${var.app_name}-codepipeline-bucket"
  acl    = "private"

  tags = {
    Environment = var.environment
  }
}
```

The concept of state is of the utmost importance within the Terraform environment. Terraform is able to inspect the current state, notice changes between the last deployment of the infrastructure and the current revision, and give the developer feedback as to what has changed or will change when a command is ran. Prior to each Terraform action, the state is refreshed against the utilized providers.

The Terraform command line exposes several key commands one must know before deploying. This project utilized the following commands:

- **init:** Initializes a terraform directory. Downloads required providers.
- **plan** Displays a list of the resources Terraform would provision if the current configuration were to be ran.
- **apply** Provision the resources defined in the configuration.
- **destroy** Destroys all previously provisioned resources by this configuration.

Another important concept to grasp is Terraform Modules. Terraform allows developers to structure resource definitions in folders, named modules. These modules can be reused throughout the entire configuration, or even shared on the internet. This allows for more concise resource definitions. In the case of this project, for instance, a database module has been created. All resources tied to the creation of an RDS are contained within this module. The module is then simply imported and configured through variables when it is provisioned for use with the server.

22.2 Deployment Walkthrough

This section will describe how the cloud infrastructure can be deployed. It will describe how to deploy infrastructure for both Auth0 and AWS. The guide assumes that the Terraform and AWS CLI's are installed.

22.2.1 Deploying to Auth0

As previously mentioned, Auth0 is responsible for authentication and authorization. Due to the limited responsibility of Auth0, very little infrastructure is provisioned.

In order to be able to deploy the infrastructure, an Auth0 account must be created. Terraform requires access to an Auth0 application with elevated privileges. This can be achieved by creating a machine-to-machine application with a client credentials grant. The generated client credentials are then passed to Terraform when provisioning the resources. In order to deploy the Auth0 infrastructure, the following steps must be performed:

1. Create an Auth0 Account. All provisioned resources are within the free tier.
2. Create a machine-to-machine application in Auth0.
3. Give the application access to the Auth0 Management API. Select all scopes.
4. Navigate to the Infrastructure folder in a CLI.
5. Navigate to the Auth0 folder.
6. Run *terraform apply*. Pass in the client ID and secret of the application that was created in step two. They can be found under the settings tab in the Auth0 console.

22.2.2 Deploying to AWS

The initial deployment of the AWS portion of the infrastructure is a three-step process. This subsection will describe how to configure the GitHub account, how to store secrets in Parameter Store, and how to deploy the actual infrastructure.

Setting up GitHub

This walkthrough assumes that a GitHub repository has already been set up, and will therefore not cover it. CodePipeline requires an access token to the GitHub API in order to fetch code from repositories. The token will also be used to provision GitHub resources. Generating an access token can be achieved by navigating to developer settings in a GitHub account. Copy the generated token and store it for the next step.

Storing Secrets in Parameter Store

Sensitive information required by either the infrastructure or the applications running on it are all stored in the AWS Parameter Store. By storing sensitive information in a secure place, like the Parameter Store, the chances of experiencing a security breach is reduced drastically.

The Parameter Store definitions are stored in a separate folder from the rest of the infrastructure. This choice was made to allow developers to not have to input all secrets every time a new revision of the infrastructure is applied. Secrets can be defined by simply passing in the variables the command line prompts when *terraform apply* is ran within the Parameter Store folder. By default, Terraform assumes that all variables should be changed when running *terraform apply*. Changing a single variable can be achieved by running *terraform apply target=aws_ssm_parameter.***${PARAMETER_NAME}**. The following list will provide a brief description of the parameters that must be passed in when provisioning system secrets:

1. Navigate to the Infrastructure folder.
2. Navigate to the *ssm* folder.
3. Run *terraform apply* in the command line.
4. The database password can be anything. A secure password is recommended.
5. *Client ID* and *Client Secret* can be found in Auth0. They can be found under the *SuperSau Server* application.
6. The *GitHub Oauth Token* variable is the access token created in Section 22.2.2.

Deploying the AWS Infrastructure

Deploying the actual AWS infrastructure is also done through Terraform. By navigating to the server folder and running *terraform apply*, everything should be provisioned automatically. All variables used within the server infrastructure has been given sensible default values, so that they do not have to be altered unless the developers requires a change.

Chapter 23

Infrastructure Discussion

This chapter will discuss some of the choices made during the cloud infrastructure development process. It will mostly detail what could have been done differently, and what should be done to further improve the end result.

23.1 AWS Educate Limitations

The AWS account used throughout the development period was a so-called AWS Educate account, intended for use by students who are learning how to use cloud providers. It offers students $30 in credits, as well as the regular free tier usage AWS offers to all new users.

Choosing an Educate account proved to be very useful to the author, as the development period saw several unexpected spikes in cloud costs. Knowing that no matter what, unexpectedly high costs would only result in a closed account provided a sense of comfort. However, Educate accounts do have some restrictions. The first, and most noticeable, is that you are constrained to only using availability zones in North Virginia, USA. Seeing as the system is intended for use in Norway, this could prove to be confusing to anyone not familiar with the project. If one were to actually deploy this application to another region, it would be as simple as changing a few lines of code in the Terraform project.

AWS Educate accounts also provides several restrictions on what resources can be used. For instance, domain name registration is entirely blocked. If one were to point a domain to the cloud infrastructure resources, it would have to be registered somewhere else. The author considered purchasing a domain for this reason, but opted to not do it. If the system were to ever be placed in a production-ready environment, adding a domain and SSL should be highly prioritized, as it drastically increases the overall security of the project.

A problem that did affect the author during development was Identity Access Management. Educate accounts are not allowed to create new user accounts with programmatic access to AWS. When using Terraform, it is considered best practice to run Terraform commands using an account with only the privileges required in order to run Terraform. However, being unable to do so, this project had to run all Terraform commands with the default AWS administrator account. If the project were to be moved over to a non-Educate account, this should be addressed. It can be solved by simply creating a new IAM account within the AWS console, and replace the credentials stored in the AWS CLI.

Another problem somewhat affecting the security of the infrastructure is the way CodePipeline is linked to GitHub. In the current infrastructure project, CodePipeline is able to access GitHub through the use of an access token. Although this access token only grants access to creating web hooks and reading repositories, it is applicable for an entire GitHub account. Having the access token leaked would not only provide attackers with access to this GitHub project, but all projects under this GitHub account. This way of connecting CodePipeline and GitHub is considered to be deprecated, and has since been improved through the use of a *CodeStar connection*, which allows a connection between a specific external source code repository and AWS resources. Once the connection has been created, it simply has to be authorized within the AWS console. However, Educate accounts were not authorized to create CodeStar connections throughout the development process. If one were to transfer away from the Educate account, the author suggests switching to CodeStar connections.

23.2 Identity and Access Management

Using Auth0 as an Identity and Access Management solution was a decision made prior to choosing a cloud provider. In hindsight, choosing the IAM solution provided by AWS, Cognito, could have been a better choice. Cognito tightly integrates with several other AWS resources, and would ensure that the entire infrastructure would be hosted within the AWS ecosystem. Furthermore, this would enable one to authorize all requests passed through the load balancer, as opposed to implementing authorization logic on every API endpoint.

The major benefit of Auth0 is the ease of use. Auth0 provides a detailed API providing access to the entire Auth0 platform, if desired. Furthermore, it provides a fully configurable and customizable hosted UI. Cognito, on the other hand, is reliant on utilizing the AWS Software Development Kit, which exposes AWS endpoints. Their hosted UI is not nearly as configurable as the one provided by Auth0. Aesthetically, the Cognito UI can appear unprofessional and insecure to a new user. Furthermore, it does not provide support for localization. If one were to utilize the Cognito hosted UI, the user interface would only be available in English. The alternative would be to implement a custom login user interface. AWS

provides Amplify, a tool for building mobile and web applications, for this very purpose.

The author found this solution to be out of scope for the current system revision. The author is not able to find any specific reason to keep identity and access-management within the AWS environment, especially when more suited alternatives exists outside of the ecosystem. Furthermore, the process of creating a custom login page could prove to be too time-consuming, and could detract from the overall time allotted for critical system components. Instead, the author chose to not depend on Auth0 too strongly. Migrating to Cognito later-on would therefore be rather simple.

Part V

Closing

Chapter 24

Project Discussion

This section provides a discussion surrounding the system as a whole, and mostly revolves around key factors that affected the process. Furthermore, it will discuss a set of functional requirements that could not be implemented within the time constraints of the project. Finally, the chapter provides a discussion surrounding potential future work and a discussion surrounding the potential benefits the system could bring to the sheep supervision process.

24.1 The Pandemic

The entirety of the project was conducted during the COVID-19 pandemic, which affected several aspects of the project. In the early stages of the project, the author and the supervisor discussed the possibility of meeting farmers and sheep observers in order to gain a greater insight into their needs and to demonstrate the developed system. However, this never came to fruition; The ongoing pandemic made meeting farmers in person unsafe. Had it not been for the domain knowledge held by the supervisor, this project would have been much harder to achieve. His extensive knowledge and experience within the sheep supervision domain allowed for the author to gain insightful knowledge as to how sheep observers would benefit from the developed system. Seeing as the mobile application is mostly targeted towards sheep observers, the supervisor was able to provide feedback on nearly every area. The author still believes that meeting sheep observers and farmers could have expedited and improved the development process, especially with regards to tailoring the system towards the needs of the ones who will be actually using it. At the current time, the system appears to fulfill most needs, but one cannot be certain until the actual users voice their opinions.

The pandemic also largely affected the communication between the supervisor and the author. In the early stages of the project, the two met in-person every week,

allowing the supervisor to test new prototypes on physical devices. As the pandemic grew worse, meetings had to be conducted remotely. This lead to the supervisor losing the ability to actually test the prototypes; Instead, the author would simply have to display them through a video feed. The core concepts within the prototypes could still be validated by the supervisor, but the act of physically experiencing them was lost. In large, the author believes this to have been inconsequential, largely thanks to supplementary usability testing. Test results indicates that the final product is user-friendly to an acceptable extent.

24.2 System Composition

An interesting position to consider when discussing the system as a whole is the choice of technologies for the backend and the mobile application. The author ultimately decided to utilize the tools he felt were most suited for the task at hand. Several other technologies and approaches could have provided similar results. In hindsight, knowing that the system could benefit from a separate website, one could have made a stronger argument for choosing React Native. The framework originated from React, a commonly used framework for creating web sites. Since being created, React Native has returned to its web roots through the framework "React Native for Web". By utilizing it, some mobile components could also have been utilized on the web site. The same logic applies to the server: Domain and infrastructure layers could nearly be copied from the mobile solution to the server.

Having finished the development of both the mobile application and the server, the author believes the aforementioned approach to provide little benefit over the chosen one. In terms of user interface reuse, very few components would be directly transferable to the web. In terms of functionality, the two system components are concerned with different things. The website would primarily be concerned with reviewing supervision data, and requires drastically different types of interaction on different devices. In large, the perceived benefit of such an approach would be a reduced amount of technologies for future developers to learn.

24.3 SQL vs NoSQL

The choice of using relational databases over non-relational ones was decided early in the project, and is possibly the technology choice one can most easily argue against. By choosing a relational approach, the application gains several benefits: Table relations, cascading deletes, improved query support and transactional consistency are all benefits only brought by relational databases. However, one can also make a sound argument for choosing NoSQL for this system; Many NoSQL solutions provide native support for state synchronization between devices. This would essentially remove the need for writing synchronization support entirely. Not only would this be a reduction in development time, but also in system complexity.

Another benefit brought by NoSQL is scalability. Relational databases are hard to scale out: Separating a relational database over several computational instances are notoriously difficult. Non-relational databases, on the other hand, can handle such difficulties with ease. In the context of this project, relational database scalability will never be a concern. The sheep supervision field is finite in size and will never scale beyond a certain point. A relational database will be able to handle a workload much greater than what is required within the context of sheep supervision.

The obvious benefit of utilizing a relational database is relations. Every entity throughout the sheep supervision domain is somehow connected; Changes within one entity can and will propagate to other entities. Deleting a farm should result in all observations of that farm being deleted as well. Relational databases allows for one to resolve such problems through the use of foreign keys and cascading deletes, and ensures the soundness of the database at any given time. By utilizing a non-relational approach, this responsibility would be transferred to the developer, increasing the overall complexity of the system and essentially negating the reduction in complexity offered by the built-in synchronization functionality.

Relations also provides a significant benefit with regards to querying data. Due to the structural integrity provided by relational databases, performing complex queries can be encapsulated entirely within SQL. By choosing a non-relational approach, this responsibility would be shifted over to the developer.

The field of sheep supervision has a ceiling cap with regards to the amount of resources that will be stored within the system. Non-relational databases allows one to easily scale the database, and would be an interesting choice if the data requirements of the system were larger. However, this is not a realistic situation within the context of this project. The author believes the benefits brought by relational databases largely outweighs the scalability benefits provided by non-relational databases. Complex querying capabilities and transactional soundness are too beneficial for the system quality as a whole. As such, he still believes the relational approach to be the correct choice at the end of the project.

24.4 Unimplemented Functional Requirements

At the time of writing, seven of the 79 function requirements listed in Section 3.4 are not entirely fulfilled. These requirements can be found in Table 24.1, and are unmet due to two different reasons. FR 5.7 and 7.9 were left unimplemented due to a lack of time. Both requirements will allow for an improved user experience, but are not required in order for the application to function as intended. The functionality for achieving FR 5.7 is available throughout the backend, but has no user interface to access it.

ID	Priority	Description
FR3.0	High	The system should allow for farmers to register their farms.
FR3.1	High	The system should allow for farmers to delete a registered farm.
FR3.2	High	The system should allow for farmers to update their registered farms.
FR5.7	Low	The mobile application should allow team owners to change the name and description of a team.
FR7.9	Low	The mobile application should allow for users to rename a downloaded map.
FR11.0	High	The system should allow for users to generate supervision reports based on supervision data at the end of a season.
FR11.1	High	Generated supervision reports should be available for download as a PDF.

Table 24.1: Unmet functional requirements.

FR 3.0, 3.1, and 3.2 are left unmet not only due to time, but also due to the original vision of the system as a whole. The author had envisioned farm management-related functionality to only be available through a website. The mobile application is primarily intended for usage by sheep observers; Requiring farmers to download the application for the sole purpose of creating, editing and deleting farms would be cumbersome. Furthermore, the author believed that the time it would take to temporarily add this functionality to the mobile application could be better spent elsewhere. As such, he decided to leave them unimplemented. The functionality is fully supported on the server; It simply needs an accompanying user interface. If need be, farms can always be added, modified and deleted through the Swagger UI described in Section 16.6 until a proper user interface is created for it.

Functionality surrounding the automatic generation of supervision reports, as listed in FR11.0 and FR11.1, could not be implemented within the time constraints of this project. Furthermore, the author believes the report generation process to be complicated enough to warrant a complex user interface requiring a larger screen size than what most mobile devices provide. Before generating a report, the author assumes that the farmer would like to review every supervision trip made throughout a season so that obvious errors could either be omitted or modified. Such work can be both precise and time-consuming, and does not lend itself well to small, mobile user interfaces; A web user interface could prove to be more beneficial, as the screen sizes are larger. As such, the process of creating a generated report is tied up to the process of creating a website.

The author also believes the process of report generation to be more reliant on feedback from project stakeholders than any other functionality the system cur-

rently provides. Whereas the creation of the server and mobile application could largely be validated through the project supervisor, the requirements of report creation cannot be solidified without having continuous communications with farmers and the governing bodies receiving the reports. Creating functionality relating to report generation without first having established the requirements of every interested stakeholder could result in functionality that could prove to be more proof-of-concept than anything else. Instead, the author chose to focus on allowing for report generation functionality to be created at a later point as easily as possible.

24.5 Distribution of Quality and Quantity

As the project progressed, the author faced the crossroads of choosing between creating new functionality and improving upon what has already been created. At the time, the author had planned to create a website tailored towards farmers and various government employees. As discussed later-on in Section 24.6.2, the web site would provide users with the ability of adding farms, generating reports and gaining a greater overview of events within a map area. Implementing this functionality would certainly increase the usefulness of the system, albeit at the cost of a reduction in quality for the rest of the system.

Through discussions with the project supervisor, the author decided to further improve existing functionality. The reasoning behind this choice mostly relates to extensibility; Having a strong foundation to build upon would allow for new functionality to be added at a more rapid pace. Ideally, one would be able to choose both quality and quantity. However, time constraints often forces one to choose one over the other. Delivering a system full of features with low quality could potentially detract from the perceived usefulness of the product. By choosing to reduce the amount of technical debt associated with the system, future development can solely focus on extending instead of refactoring. The time gained by not implementing new functionality was instead spent on improving the architecture, infrastructure and documentation. In hindsight, the author believes this to have been the better choice between the two. He is hopeful for the future of the system; As such, preparing it for further development can be considered to be a sound choice. A well-defined architecture with extensive documentation will allow future developers to quickly gain an understanding of the system, and extend it in any way necessary. Choosing to implement new functionality would have provided more immediate benefits. However, the author believes such actions to be harmful to the project in the long run, as technical debt could accrue to the extent that the system would require major refactoring.

24.6 Future work

This project had a strict deadline on June 1st, 2021. As such, the author had to come to terms with limiting the scope of the project. This section will describe functionality the author had envisioned as being an integral part of the system, but had to be omitted due to time constraints.

24.6.1 Usability Testing Farmers and Sheep Observers

The entirety of this project was conducted during the COVID-19 pandemic. As such, some precautions had to be taken. In terms of working on the project, the pandemic proved to be largely irrelevant. It did, however, significantly impact the possibility of meeting sheep observers and farmers. As such, every usability test had to performed on subjects with no relation to the sheep supervision process. In order to gain a greater and more accurate understanding of the usefulness of the developed system, usability tests with farmers and sheep observers should be conducted. The usability tests performed throughout this project can be considered to provide some context with regards to the potential benefits of a digital supervision system, but cannot be considered to be absolute before the actual users can voice their opinions.

24.6.2 Creating a Web Site

Throughout the master thesis, the author and his supervisor discussed the creation of a website tailored towards the farmers that utilize the system. The web site was intended to be based around an interactive map that allowed farmers to view sheep supervision trips within a specific map area. This functionality could have allowed for farmers to gain a greater understanding of the current whereabouts of grazing sheep, and a greater overview of any potential threats within those areas.

Functionality relating to the creation and modification of farms were to be placed within the web site. If one were to move the responsibility of farm management over to the website, farmers would no longer need to download the mobile application; Management-related tasks could be performed solely through the website. The backend already fully supports farm creation, modification and deletion, but are not currently available through any user interface. Farm-related functionality must be completed in order to fulfill the unmet functional requirements listed in Table 24.1. If they are not made available through the website, they must be made available through the mobile application.

Report generation was another set of functionality intended solely for the web application. The choice of placing it within the domain of the website is two-fold: It only relates to farmers, and it is an involved process. A farmer cannot simply assume that all data provided throughout a grazing season is valid. Every trip must be reviewed to some extent, and even possibly modified or removed. The

author believes such concerns to be better suited for a larger user interface, making the website a perfect fit; The average computer display is much larger than the average smartphone. Furthermore, data manipulation is a task much better suited for computers. Displaying and altering a large set of fields and columns on a mobile user interface can become confusing due to the size of the screen.

24.6.3 Integrating With External Actors

This project is not the only one concerned with the supervision and general well-being of sheep; Integrating this system with other systems could prove to bring several benefits. Rovbase and Skandobs are primarily concerned with keeping track of predator observations throughout Norway and Sweden. Any observations of predators registered within this system could automatically be sent over to either party, and could potentially increase the overall awareness of the whereabouts of predators in Norway and Sweden. Furthermore, integrating with the aforementioned observation systems could allow for farmers to be automatically notified whenever a new predator observation is registered in their systems. Observations within Rovbase and Skandobs could possibly even be displayed on the maps used within both the map application and the website, allowing for farmers to more easily adapt their supervision frequency.

Integrating the system with electronic tracking services could also prove to be very beneficial for sheep observers and farmers alike. The mobile application could display the last known whereabouts of tracked sheep, which would aid sheep observers when they are planning a supervision trip. Furthermore, farmers could benefit from viewing live tracking data through the use of the web site described in Section 24.6.2, allowing farmers to gain a greater understanding of the current and past whereabouts of their sheep.

Sheep observers should notify Statens Naturoppsyn any time the death of a sheep could be possibly linked to a predator. At the time of writing, this process is conducted manually. Allowing the server to automatically notify SNO of any sheep deaths linked to predators would reduce the workload of the sheep observer, and possibly even reduce the chance of human error.

24.6.4 Report Generation

When the grazing season ends, farmers are expected to create and send a sheep supervision report to several governing bodies, a process currently being conducted manually. The project originally intended to create functionality for automatically generating reports once the grazing season ended, but ultimately had to be cut due to a lack of time. If the project were to ever be continued, report generation should be a top priority. The author has imagined this functionality to be available through the use of a website, as it would allow for one to more easily modify the contents of the report to generate. A farmer should ideally be able to modify or

omit any trip data they see fit, which could prove to become disorganized and confusing on smaller user interfaces.

At the time of writing, the mobile application is able to generate every type of data the government is interested in receiving, allowing for a large foundation of data to choose from when generating reports. Furthermore, the server provides extensive querying and filtering capabilities. Creating functionality for report generation simply requires one to splice them together, and allow for the functionality to be accessed through a user interface. In order for the generated report to benefit as many stakeholders as possible, the development process surrounding the functionality should strive to include the stakeholders to the largest extent possible.

24.7 Potential Benefits of Digital Supervision

At the beginning of the project, the author expected to be able to perform usability tests on actual farmers at some point during the master thesis. Unfortunately, this never came to fruition due to the ongoing COVID-19 pandemic. Instead, subjects unrelated to the supervision process had to be chosen. As such, their experiences with the system might not be representative for farmers and sheep observers. The feedback received from usability testing mostly reflected the feedback from the supervisor, granting it some validity due to the extensive experience the supervisor has with the current supervision process. Throughout usability tests, prototypes and discussions with the project supervisor, the author uncovered numerous potential benefits one could gain by conducting supervision trips through the use of the developed system. This section will separate the potential benefits into three categories, each corresponding to a research question.

What is to be gained from collecting structured sheep supervision data?

One of the biggest benefits a digital supervision solution can provide is data consistency. Whereas the current method of performing supervision trips provides loose guidelines for data collection and supervision details, this system forces every user to provide the same data, thus standardizing the collected data and the supervision process. A consistent set of data lays the groundwork for performing data analysis at a later point. For instance, analyzing the movement of herds of sheep for abnormal movement patterns might allow farmers to gain a greater understanding of whether something is amiss. Furthermore, having a set of historical sheep locations can allow sheep observers to optimize their supervision routes to increase supervision efficiency. This benefit could be further extended by combining application data and data generated from electronic tracking devices.

Digitally storing supervision data will also allow for actors and external services to access it more easily, enabling actors such as Mattilsynet and the county governor to continuously monitor supervision data. Furthermore, third-party services concerned with managing other aspects of tending to sheep can seamlessly poll for

updated sheep data. Livestock management services can for instance periodically check for dead sheep observations and automatically update their state.

Standardizing the set of data a sheep observer must register might also alleviate problems surrounding a lack of data when submitting seasonal reports, especially with regards to monetary reimbursements for sheep lost to predators. Every farmer will have submitted data in the same format, thus allowing for one to more easily make a decision that aligns with previous results in cases with similar data. Furthermore, a larger set of data to analyze will allow Mattilsynet and the county governor to gain a greater understanding of whether the performed supervisory work is up to standards.

In what ways can digitalized sheep supervision affect the well-being of grazing sheep?

The well-being of sheep grazing on outlying fields can be affected in a multitude of ways by utilizing this system, with the most transparent one being an increase in information sharing between farmers and sheep observers. An isolated or ill sheep can be registered by any observer within a supervision team, in turn allowing the relevant farmer to act on the observation.

Furthermore, making observation data immediately available to any farmers and observers within a supervision team will allow for sheep observers to plan their trip routes around points where previous sheep observations have occurred. The process of doing so can ensure that every sheep receives more supervision, and also improve the efficiency of the route of the observer.

One could also argue that improving the efficiency of the overall supervision process might allow sheep supervision to occur more frequently, or in the least cover more ground throughout a single trip. The author believes this to be largely irrelevant, as the increase in efficiency is akin to a decrease in tediousness more than anything else.

The process of registering "other" observations can also increase the well-being of grazing sheep to some extent. At the time of writing, farmers already employ a tactic of increasing supervision trips in areas where predators have been sighted. The knowledge of predator sightings can be more easily shared by utilizing the application, thus allowing farmers to be more aware of when additional supervision is required. This benefit can be further increased by integrating the server with Rovbase or Skandobs, third-party services for registering predator sightings. Any predator observations registered within the application could be automatically sent to the aforementioned services, removing the need of registering said sightings within Rovbase or Skandobs manually. Furthermore, any sightings registered within the Rovbase system could be displayed within this system, allowing for a greater overview of predator sightings.

On the topic of third-party services, integrating the application with systems for electronic sheep tracking can further increase the efficiency of the observation route, whilst also ensuring that as many sheep as possible are checked up on in every supervision trip.

What potential benefits can sheep observers and farmers gain by utilizing digital supervision solutions?

Digitally storing sheep supervision details can allow farmers to gain a greater overview of where their sheep are at any given time. Furthermore, digital storage completely removes the risk of losing sheep supervision notes. Lost supervision notes could not only lead to fines due to breaching the guidelines of sheep supervision, but it could also impact the user in terms of monetary compensation for sheep deaths caused by predators. To build upon this, digital supervision registration allows one to consolidate all supervision note-taking into the same process. The application can allow an observer to automatically store geospatial data about what areas a trip covered, a process previously dependent on external systems.

Furthermore, conducting sheep supervisions through the application can allow the sheep observers to register observations at a more rapid pace. The mobile application is capable of automatically retrieving the time and location of both trips and observations, completely eliminating the need to manually register them. Additionally, the application may reduce the amount of inaccurate observation data, especially with regards to sheep ties and locations. A set of coordinates are very precise, and writing the wrong number can easily occur. The application negates this risk of human error entirely, as coordinates are automatically retrieved by the application. With manual registration, the sheep observer has to memorize the number of ties per color in their head while counting; By using the mobile application, the responsibility of remembering the count is transferred to a computer.

Communication between farmers and sheep observers could also benefit from this system through the concept of teams. The supervision teams the system provides can greatly alleviate the amount of communication between farmers and observers across several different farms. Phone calls, text messages and emails can be avoided by simply checking the trips for a specific team. The same logic applies for "other" observations. Instead of manually notifying all nearby farmers of predator sightings, farmers can instead simply check the system.

The most interesting potential benefits of the created system, however, are the ones further extensions can introduce. Section 24.6 discusses several extensions to the system, allowing for potential benefits for many different stakeholders. The aforementioned extension points are the main driver behind the system architecture: Extensibility. Every component has an extensible architecture, allowing one to easily add and remove additional system components.

Chapter 25

Conclusion

This project explored the concept of digitalizing the sheep observation documentation process during government-mandated sheep supervision trips to discover what benefits it could provide over the currently analog process. The developed system consists of a mobile application for performing sheep supervision trips, a server for centrally storing data, and a cloud infrastructure supporting the system. In large, the system serves as a base on which one can extract additional benefits through extension; Instead of quickly creating functionality of sub-par quality, development time was spent refining the system architecture and documentation.

The usability tests conducted throughout the thesis proved that the system is easy to use and understand. Testing could not be performed on the actual users of the system due to COVID-19. As such, perceived benefits cannot be viewed as absolute. If system development is continued, it would be wise to conduct interviews and usability tests on the intended system users. In order for the system to function as envisioned, functionality for report generation and farm mutation should be implemented. Furthermore, a website tailored towards farmers could be beneficial.

By creating prototypes and conducting formal and informal usability tests, the author discovered the benefits a digitalized supervision process could entail. These findings were further validated by the supervisor, a seasoned sheep observer. System usage could increase and standardize supervision trip data. Centrally storing data can make it more accessible and reduce the chance of data loss. It can also serve as a foundation for data analysis. The mobile application can facilitate the registration process by automatically generating required supervision data, lowering the chance of human error. An increased amount of data can in turn benefit system stakeholders and the well-being of sheep, by allowing for optimized supervision routes and improved communications. Implementing the functionality suggested in Section 24.6 could yield additional benefits; To accommodate such functionality, the system has been designed with extensibility in mind.

Bibliography

[1] Mattilsynet. (Mar. 2020). Årsrapport 2019 - mattilsynet, [Online]. Available: https://www.mattilsynet.no/om_mattilsynet/aarsrapport_2019__mattilsynet.38708/binary/%C3%85rsrapport%202019%20-%20Mattilsynet (Accessed 6 Nov. 2020).

[2] Rovbase. (2020). Erstatning for sau (Norge 2019), [Online]. Available: https://rovbase.no/erstatning/sau (Accessed 7 Nov. 2020).

[3] Lovdata. (Feb. 2005). Forskrift om velferd for småfe, [Online]. Available: https://lovdata.no/dokument/SF/forskrift/2005-02-18-160 (Accessed 6 Nov. 2020).

[4] Statistisk Sentralbyrå. (2020). Land use and land cover, [Online]. Available: https://www.ssb.no/en/arealstat (Accessed 30 Mar. 2021).

[5] J. Schärer. (Dec. 2016). Norge - et utmarksland, [Online]. Available: https://www.nibio.no/nyheter/norge--et-utmarksland (Accessed 5 May 2021).

[6] Norsk Sau og Geit. (n.d). Verdien av beitebruk, [Online]. Available: https://www.nsg.no/beitebruk/beiterett/verdisetting-av-beitebruk/verdien-av-utmarksbeite/ (Accessed 11 Mar. 2021).

[7] G. Austrheim, L. Asheim, G. Bjarnason, J. Feilberg, A. M. Fosaa, O. Holand, K. Høegh, I. Jónsdóttir, B. Magnusson, L. Mortensen, A. Mysterud, E. Olsen, A. Skonhoft, G. Steinheim, and A. Thorhallsdottir, "Sheep grazing in the north atlantic region: A long term perspective on management, resource economy and ecology," *Vitenskapsmuseet, Rapport Zoology Series*, vol. 2008, 3, Jan. 2008.

[8] Norsk Sau og Geit. (n.d). Bjelleslips - kodemerking for lammetall på beite, [Online]. Available: https://www.nsg.no/a-a/merking-av-smafe/bjelleslips/ (Accessed 28 Nov. 2020).

[9] Norsk Sau og Geit. (Jun. 2020). Slips og fargekoder, [Online]. Available: https://www.nsg.no/Oppland/slips-og-fargekoder (Accessed 28 Nov. 2020).

[10] Regjeringen. (n.d). Rovdyr i noreg, [Online]. Available: https://www.regjeringen.no/no/tema/klima-og-miljo/naturmangfold/innsiktsartikler-naturmangfold/rovvilt-og-rovviltforvaltning/id2076779/ (Accessed 5 May 2021).

[11] Beitesnap. (n.d). En revolusjonerende beiteapp! [Online]. Available: https://www.beitesnap.no/ (Accessed 24 Nov. 2020).

[12] (Sep. 2017). Nyttig app til beitesesongen, [Online]. Available: https://www.bondevennen.no/aktuelt/nyttig-app-til-beitesesongen/ (Accessed 24 Nov. 2020).

[13] Rovdata and Naturvårdsverket. (n.d). Skandobs, [Online]. Available: https://www.skandobs.no/#showAbout (Accessed 24 Nov. 2020).

[14] Findmy. (n.d). Produkt, [Online]. Available: https://www.findmy.no/nb/produkt (Accessed 7 Apr. 2021).

[15] Telespor. (n.d). Telespor, [Online]. Available: https://telespor.no/ (Accessed 7 Apr. 2021).

[16] Nofence. (n.d). Nofence, [Online]. Available: https://www.nofence.no/ (Accessed 7 Apr. 2021).

[17] Smartbjella. (n.d). Smartbjella, [Online]. Available: https://smartbjella.no/ (Accessed 7 Apr. 2021).

[18] Norsk Sau og Geit. (n.d). Verdisatser for småfe, [Online]. Available: https://www.nsg.no/a-a/okonomi/verdisatser/ (Accessed 11 May 2021).

[19] Findmy. (n.d). Bestill e-bjella model2 nå! [Online]. Available: https://www.findmy.no/shop/ (Accessed 17 May 2021).

[20] Telespor. (n.d). Radiobjella, [Online]. Available: https://nettbutikk.telespor.no/categories/radiobjella (Accessed 17 May 2021).

[21] Nofence. (2021). Gjeldende prismodell for bruk, [Online]. Available: https://www.nofence.no/fakturainfo (Accessed 17 May 2021).

[22] Smartbjella. (n.d). Bestill - smartbjella sporing, [Online]. Available: https://smartbjella.no/butikk/ (Accessed 17 May 2021).

[23] Kartverket. (2020). Hvor?-appen, [Online]. Available: https://www.kartverket.no/til-lands/kart/hvor-appen (Accessed 25 Nov. 2020).

[24] Ut.no. (n.d). Ut.no, [Online]. Available: https://apps.apple.com/no/app/ut-no/id510575024 (Accessed 25 Nov. 2020).

[25] I. L. Hjelmeland. (n.d). Topokart - ditt turkart, [Online]. Available: https://apps.apple.com/no/app/topokart-ditt-turkart/id1090025147 (Accessed 25 Nov. 2020).

[26] Norgeskart. (n.d). Norgeskart, [Online]. Available: https://apps.apple.com/no/app/norgeskart/id727189627?l=nb (Accessed 27 May 2021).

[27] Ture Apps AS. (n.d). Norgeskart outdoors, [Online]. Available: https://ture.no/en/norgeskart-english/ (Accessed 25 Nov. 2020).

[28] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd. Addison-Wesley Professional, 2012, ISBN: 0321815734.

[29] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Comput. Surv.*, vol. 15, no. 4, pp. 287–317, Dec. 1983, ISSN: 0360-0300. DOI: 10.1145/289.291.

[30] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed. USA: Prentice Hall PTR, 2008, ISBN: 0132350882.

[31] H. Kniberg and M. Skarin, *Kanban and Scrum - Making the Most of Both*. Lulu.com, 2010, ISBN: 0557138329.

[32] Trello. (n.d). What is trello? [Online]. Available: https://help.trello.com/article/708-what-is-trello (Accessed 7 Apr. 2021).

[33] Auth0. (n.d). Auth0, [Online]. Available: https://auth0.com/ (Accessed 21 May 2021).

[34] Microsoft, *Visual studio code - code editing. redefined*, version 1.51.0, Oct. 2020. [Online]. Available: https://code.visualstudio.com/ (Accessed 21 May 2021).

[35] GitHub. (n.d). Github: Where the world builds software, [Online]. Available: https://github.com/ (Accessed 21 May 2021).

[36] Apple. (n.d). Ios 14 - apple developer, [Online]. Available: https://developer.apple.com/ios/ (Accessed 29 May 2021).

[37] Google. (n.d). What is android, [Online]. Available: https://www.android.com/what-is-android/ (Accessed 29 May 2021).

[38] StatCounter. (2020). Mobile operating system market share worldwide, [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-200901-202010 (Accessed 3 Dec. 2020).

[39] StatCounter. (2020). Mobile operating system market share norway, [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/norway/#monthly-200901-202010 (Accessed 3 Dec. 2020).

[40] StatCounter. (2021). Desktop vs mobile vs tablet market share worldwide, [Online]. Available: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide/#monthly-200901-202104 (Accessed 29 May 2021).

[41] Ionic. (n.d). Open-source ui toolkit to create your own mobile or desktop apps, [Online]. Available: https://ionicframework.com/docs (Accessed 22 May 2021).

[42] Facebook. (n.d). React native - learn once, write anywhere, [Online]. Available: https://reactnative.dev/ (Accessed 21 May 2021).

[43] The Flutter Team, *Flutter - beautiful native apps in record time*, version 2.0.2, Mar. 2021. [Online]. Available: https://flutter.dev/ (Accessed 21 May 2021).

[44] Google, *Dart programming language | dart*, version 2.12.1, Mar. 2021. [Online]. Available: https://dart.dev/ (Accessed 23 May 2021).

[45] T. Sneath. (May 2021). Announcing flutter 2.2 at google i/o 2021, [Online]. Available: https://medium.com/flutter/announcing-flutter-2-2-at-google-i-o-2021-92f0fcbd7ef9 (Accessed 22 May 2021).

[46] Microsoft. (n.d). Xamarin | open-source mobile app platform for .net, [Online]. Available: https://dotnet.microsoft.com/apps/xamarin (Accessed 21 May 2021).

[47] Facebook. (n.d). Who's using react native? - react native, [Online]. Available: https://reactnative.dev/showcase (Accessed 22 May 2021).

[48] G. Peal. (Jun. 2018). React native at airbnb, [Online]. Available: https://medium.com/airbnb-engineering/react-native-at-airbnb-f95aa460be1c (Accessed 5 Dec. 2020).

[49] G. Peal. (Jun. 2018). React native at airbnb: The technology, [Online]. Available: https://medium.com/airbnb-engineering/react-native-at-airbnb-the-technology-dafd0b43838 (Accessed 5 Dec. 2020).

[50] G. Peal. (Jun. 2018). Sunsetting react native, [Online]. Available: https://medium.com/airbnb-engineering/sunsetting-react-native-1868ba28e30a (Accessed 5 Dec. 2020).

[51] F. Thawar. (Jan. 2020). React native is the future of mobile at shopify, [Online]. Available: https://shopify.engineering/react-native-future-mobile-shopify (Accessed 5 Dec. 2020).

[52] The Flutter Team. (n.d). Showcase - flutter, [Online]. Available: https://flutter.dev/showcase (Accessed 22 May 2021).

[53] C. Sprague and L. McKenzie. (Sep. 2020). Ebay motors, accelerating with flutter, [Online]. Available: https://tech.ebayinc.com/product/ebay-motors-accelerating-with-fluttertm/ (Accessed 5 Dec. 2020).

[54] H. Wang. (Jan. 2019). Competing frameworks: Alibaba puts sdk flutter to the test, [Online]. Available: https://medium.com/hackernoon/competing-frameworks-alibaba-puts-sdk-flutter-to-the-test-88eb8cf1f35a (Accessed 5 Dec. 2020).

[55] Microsoft. (n.d). Xamarin customer showcase, [Online]. Available: https://dotnet.microsoft.com/apps/xamarin/customers (Accessed 22 May 2021).

[56] Stack Overflow. (May 2020). Developer survey, [Online]. Available: https://insights.stackoverflow.com/survey/2020 (Accessed 14 Mar. 2021).

[57] A. Bjørn Hansen, C. Rieger, T.-M. Grønli, T. A. Majchrzak, and G. Ghinea, "An empirical investigation of performance overhead in cross-platform mobile development frameworks," Jun. 2020. DOI: https://doi.org/10.1007/s10664-020-09827-6.

[58] M. Willocx, J. Vossaert, and V. Naessens, "Comparing performance parameters of mobile app development strategies," in *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2016, pp. 38–47. DOI: 10.1109/MobileSoft.2016.028.

[59] L. Corbalán, P. Thomas, L. Delia, G. Cáseres, J. Fernandez Sosa, F. Tesone, and P. Pesado, "A study of non-functional requirements in apps for mobile devices," in. Jul. 2019, pp. 125–136, ISBN: 978-3-030-27712-3. DOI: 10.1007/978-3-030-27713-0_11.

[60] B. De Coninck. (Dec. 2019). Flutter versus other mobile development frameworks: A ui and performance experiment. part 2, [Online]. Available: https://blog.codemagic.io/flutter-vs-android-ios-xamarin-reactnative/ (Accessed 5 Dec. 2020).

[61] D. Tuppeny, *Flutter - visual studio marketplace*, version 3.17.1, Dec. 2020. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=Dart-Code.flutter (Accessed 21 May 2021).

[62] Google, *Meet android studio | android developers*, version 4.1, Aug. 2020. [Online]. Available: https://developer.android.com/studio/intro (Accessed 21 May 2021).

[63] Apple, *Xcode*, version 12.4, Jan. 2021. [Online]. Available: https://developer.apple.com/documentation/xcode/ (Accessed 21 May 2021).

[64] D. Tuppeny, *Dart - visual studio marketplace*, version 3.17.1, Dec. 2020. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=Dart-Code.dart-code (Accessed 21 May 2021).

[65] W. Denniss, Google, J. Bradley, and Ping Identity, "Oauth 2.0 for native apps," RFC Editor, RFC 6749, Oct. 2017. [Online]. Available: https://tools.ietf.org/html/rfc8252.

[66] The Flutter Team. (n.d). Inside flutter, [Online]. Available: https://flutter.dev/docs/resources/inside-flutter#aggressive-composability (Accessed 2 Apr. 2021).

[67] *Background_locator | flutter package*, version 1.2.2+1, Aug. 2020. [Online]. Available: https://pub.dev/packages/background_locator (Accessed 29 May 2021).

[68] M. Bui, *Flutter_appauth | flutter package*, version 0.9.2+6, Oct. 2020. [Online]. Available: https://pub.dev/packages/flutter_appauth (Accessed 21 May 2021).

[69] The Flutter Team. (n.d). Using packages, [Online]. Available: https://flutter.dev/docs/development/packages-and-plugins/using-packages (Accessed 14 Mar. 2021).

[70] R. Rousselet, *Provider*, version 4.3.3, Jan. 2021. [Online]. Available: https://pub.dev/documentation/provider/4.3.3/ (Accessed 2 Apr. 2021).

[71] The Flutter Team. (n.d). List of state management approaches, [Online]. Available: https://flutter.dev/docs/development/data-and-backend/state-mgmt/options#provider (Accessed 2 Apr. 2021).

[72] J. Ryan, *Flutter_map | flutter package*, version 0.11.0, Jan. 2021. [Online]. Available: https://pub.dev/packages/flutter_map (Accessed 21 May 2021).

[73] S. Binder, *Moor | flutter package*, version 3.4.0, Oct. 2020. [Online]. Available: https://pub.dev/packages/moor (Accessed 21 May 2021).

[74] A. R. Tekartik, *Sqflite | flutter package*. [Online]. Available: https://pub.dev/packages/sqflite (Accessed 21 May 2021).

[75] ReactiveX, *Rxdart | flutter package*, version 0.24.1, May 2020. [Online]. Available: https://pub.dev/packages/rxdart (Accessed 21 May 2021).

[76] Flutter China, *Dio | flutter package*, version 3.0.1, Sep. 2019. [Online]. Available: https://pub.dev/packages/dio (Accessed 21 May 2021).

[77] G. Saprykin, *Flutter_secure_storage | flutter package*, version 3.3.5, Oct. 2020. [Online]. Available: https://pub.dev/packages/flutter_secure_storage (Accessed 21 May 2021).

[78] Appearance, *Camerawesome | flutter package*, version 0.1.2+1, Nov. 2020. [Online]. Available: https://pub.dev/packages/camerawesome (Accessed 21 May 2021).

[79] Baseflow, *Geolocator | flutter package*, version 6.2.1, Feb. 2021. [Online]. Available: https://pub.dev/packages/geolocator (Accessed 21 May 2021).

[80] J. Palermo. (Jul. 2008). The onion architecture: Part 1, [Online]. Available: https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/ (Accessed 4 Apr. 2021).

[81] Microsoft. (Jul. 2017). The model-view-viewmodel pattern, [Online]. Available: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm (Accessed 4 Apr. 2021).

[82] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, ser. Addison-Wesley Professional Computing Series. Pearson Education, 1994, ISBN: 9780321700698.

[83] M. Fowler. (Jan. 2004). Inversion of control containers and the dependency injection pattern, [Online]. Available: https://martinfowler.com/articles/injection.html (Accessed 5 Apr. 2021).

[84] Oracle. (n.d). Core j2ee patterns - data access object, [Online]. Available: https://www.oracle.com/java/technologies/dataaccessobject.html (Accessed 9 Apr. 2021).

[85] E. Hieatt and R. Mee. (2002). Repository, [Online]. Available: https://martinfowler.com/eaaCatalog/repository.html (Accessed 4 Apr. 2021).

[86] M. Fowler. (Nov. 2016). ValueObject, [Online]. Available: https://martinfowler.com/bliki/ValueObject.html (Accessed 8 Apr. 2021).

[87] M. Safyan. (n.d). Singleton anti-pattern, [Online]. Available: https://www.michaelsafyan.com/tech/design/patterns/singleton (Accessed 8 Apr. 2021).

[88] Google. (n.d). Firebase realtime database, [Online]. Available: https://firebase.google.com/docs/database (Accessed 24 May 2021).

[89] Google. (n.d). Cloud firestore | firebase, [Online]. Available: https://firebase.google.com/docs/firestore (Accessed 24 May 2021).

[90] Google, *Build_runner | flutter package*, version 1.11.1, Feb. 2021. [Online]. Available: https://pub.dev/packages/build_runner (Accessed 31 May 2021).

[91] M. Fowler. (Jun. 2015). Yagni, [Online]. Available: https://martinfowler.com/bliki/Yagni.html (Accessed 4 Apr. 2021).

[92] The Flutter Team, *Path_provider | flutter package*, version 1.6.27, Jan. 2021. [Online]. Available: https://pub.dev/packages/path_provider (Accessed 21 May 2021).

[93] The Flutter Team, *Mockito | flutter package*, version 4.1.4, Jan. 2021. [Online]. Available: https://pub.dev/packages/mockito (Accessed 20 May 2021).

[94] Smartbear. (n.d). Api documentation & design tools for teams | swagger, [Online]. Available: https://swagger.io/ (Accessed 21 May 2021).

[95] J. Brooke, "Sus: A quick and dirty usability scale," *Usability Eval. Ind.*, vol. 189, Nov. 1995. DOI: 10.1201/9781498710411-35.

[96] C. Sells. (Mar. 2021). What's new in flutter 2, [Online]. Available: https://medium.com/flutter/whats-new-in-flutter-2-0-fe8e95ecc65 (Accessed 31 May 2021).

[97] The Flutter Team. (May 2019). Automatic/scalable shader warm-up, [Online]. Available: https://github.com/flutter/flutter/issues/32170 (Accessed 4 Apr. 2021).

[98] SQLite Consortium, *Sqlite home page*. [Online]. Available: https://www.sqlite.org/index.html (Accessed 21 May 2021).

[99] The PostgreSQL Global Development Group. (n.d). Appendix D. SQL conformance, [Online]. Available: https://www.postgresql.org/docs/current/features.html (Accessed 11 Apr. 2021).

[100] MariaDB Foundation, *Mariadb foundation - mariadb.org*. [Online]. Available: https://mariadb.org/ (Accessed 21 May 2021).

[101] Docker Inc. (n.d). Docker overview, [Online]. Available: https://docs.docker.com/get-started/overview/ (Accessed 11 Apr. 2021).

[102] Docker Inc. (n.d). Overview of docker compose, [Online]. Available: https://docs.docker.com/compose/ (Accessed 11 Apr. 2021).

[103] JetBrains. (n.d). Comparison to java, [Online]. Available: https://kotlinlang.org/docs/comparison-to-java.html#what-kotlin-has-that-java-does-not (Accessed 11 Apr. 2021).

[104] OpenJS Foundation, *About | node.js*, version 15.1.0, Nov. 2020. [Online]. Available: https://nodejs.org/en/about/ (Accessed 24 May 2021).

[105] B. Ray, D. Posnett, P. Devanbu, and V. Filkov, "A large-scale study of programming languages and code quality in github," *Commun. ACM*, vol. 60, no. 10, pp. 91–100, Sep. 2017, ISSN: 0001-0782. DOI: 10.1145/3126905.

[106] Z. Gao, C. Bird, and E. T. Barr, "To type or not to type: Quantifying detectable bugs in javascript," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 758–769. DOI: 10.1109/ICSE.2017.75.

[107] Microsoft, *Typescript: Typed javascript at any scale*, version 4.0.5, Oct. 2020. [Online]. Available: https://www.typescriptlang.org/ (Accessed 24 May 2021).

[108] OpenJS Foundation. (n.d). Express - node.js web application framework, [Online]. Available: https://expressjs.com/ (Accessed 21 May 2021).

[109] NestJS, *Nestjs - a progressive node.js framework*, version 7.5.1, Nov. 2020. [Online]. Available: https://nestjs.com/ (Accessed 21 May 2021).

[110] TypeORM, *Typeorm - amazing orm for typescript and javascript (es7, es6, es5). supports mysql, postgresql, mariadb, sqlite, ms sql server, oracle, websql databases. works in nodejs, browser, ionic, cordova and electron platforms*. Version 0.2.32, Mar. 2021. [Online]. Available: https://typeorm.io/ (Accessed 21 May 2021).

[111] Axios, *Axios*, version 0.21.1, Dec. 2020. [Online]. Available: https://axios-http.com/ (Accessed 21 May 2021).

[112] Amazon Web Services, *Aws sdk for javascript*, version 2.872.0, Mar. 2021. [Online]. Available: https://aws.amazon.com/sdk-for-javascript/ (Accessed 21 May 2021).

[113] node-cache, *Node-cache/node-cache: A node internal (in-memory) caching module*, version 5.1.2, Jul. 2020. [Online]. Available: `https://github.com/node-cache/node-cache` (Accessed 21 May 2021).

[114] J. Hanson, *Passport.js*, version 0.4.1, Sep. 2019. [Online]. Available: `https://www.passportjs.org/` (Accessed 21 May 2021).

[115] OpenJS Foundation, *Eslint - pluggable javascript linter*, version 7.12.1, Oct. 2020. [Online]. Available: `https://eslint.org/` (Accessed 21 May 2021).

[116] Prettier, *Prettier - opinionated code formatter*, version 2.1.2, Sep. 2020. [Online]. Available: `https://prettier.io/` (Accessed 21 May 2021).

[117] M. Fowler. (n.d). Data transfer object, [Online]. Available: `https://martinfowler.com/eaaCatalog/dataTransferObject.html` (Accessed 13 May 2021).

[118] NestJS. (n.d). Modules, [Online]. Available: `https://docs.nestjs.com/modules` (Accessed 1 May 2021).

[119] TypeStrong, *Home | typedoc*, version 0.20.36, Apr. 2021. [Online]. Available: `http://typedoc.org/` (Accessed 25 May 2021).

[120] Amazon Web Services. (n.d). Amazon s3, [Online]. Available: `https://aws.amazon.com/s3/` (Accessed 25 May 2021).

[121] Facebook, *Jest - delightful javascript testing*, version 26.6.3, Nov. 2020. [Online]. Available: `https://jestjs.io/` (Accessed 27 May 2021).

[122] K. Kabra, *Ts-jest: A jest transformer with source map support that lets you use jest to test projects written in typescript*, version 26.4.3, Oct. 2020. [Online]. Available: `https://github.com/kulshekhar/ts-jest` (Accessed 27 May 2021).

[123] Sinon.JS, *Sinon.js - standalone test fakes, spies, stubs and mocks for javascript. works with any unit testing framework.* Version 10.0.0, Mar. 2021. [Online]. Available: `https://sinonjs.org/` (Accessed 27 May 2021).

[124] Vision Media, *Supertest: Super-agent driven library for testing node.js http servers using a fluent api*, version 6.0.0, Oct. 2020. [Online]. Available: `https://github.com/visionmedia/supertest` (Accessed 27 May 2021).

[125] Amazon Web Services. (n.d). Auto scaling groups, [Online]. Available: `https://docs.aws.amazon.com/autoscaling/ec2/userguide/AutoScalingGroup.html` (Accessed 4 Mar. 2021).

[126] Amazon Web Services. (n.d). Aws cloudformation - infrastructure as code & aws resource provisioning, [Online]. Available: `https://aws.amazon.com/cloudformation/` (Accessed 25 May 2021).

[127] Amazon Web Services. (n.d). Aws cloud development kit, [Online]. Available: `https://aws.amazon.com/cdk/` (Accessed 13 Apr. 2021).

[128] HashiCorp. (n.d). Introduction to terraform, [Online]. Available: https://www.terraform.io/intro/index.html (Accessed 13 Apr. 2021).

[129] R. Wang and A. Mishra. (Jul. 2020). Cdk for terraform: Enabling python & typescript support, [Online]. Available: https://www.hashicorp.com/blog/cdk-for-terraform-enabling-python-and-typescript-support (Accessed 30 May 2021).

[130] Pulumi. (n.d). Pulumi - modern infrastructure as code, [Online]. Available: https://www.pulumi.com/ (Accessed 30 May 2021).

Appendix A

Usability Testing Script

The following pages describes the script the author used when performing usability testing. It outlines the information provided to the user surrounding the sheep supervision process, usability testing and the system usability scale. Actual email addresses have been removed from the transcript, and are instead replaced with placeholder values.

1 An Introduction to Sheep Supervision

Norwegian sheep graze unsupervised on outlying fields throughout the warmer months of the year. In order to ensure animal wellness, farmers are required to check up on them several times a week. Farmers usually solve this problem by teaming up with other farmers, and sharing the responsibility of performing sheep supervision. For every sheep observation, the observer will note down the amount of sheep in the observation, their geographical location and the number of sheep separated by wool color. Adult female sheep will have a color tie placed around their neck, denoting the amount of lamb they originally had when leaving the farm. Furthermore, bi-colored strips attached to the ear of the sheep indicates the farm they belong to. If possible, the observer should attempt to register this information as well. In addition to registering sheep observations, observers should also strive to note down dead sheep, broken equipment and predator sightings. The information gathered throughout the supervision trips is condensed into a report at the end of the season, and is sent to the appropriate authorities to prove the well-being of the sheep, and possibly receive reimbursement for dead sheep.

At the time of writing, registering supervision details are most commonly performed through the use of pen and paper. This solution aims to digitalize the supervision registration process through the use of smartphones in order to improve the process of sheep supervision. By digitalizing the process, this project aims to not only simplify the data collection process, but also increase the amount and quality of data being collected.

2 What is Usability Testing?

Usability testing is the process of verifying the effectiveness and perceived ease of use of a product. In the case of this project, the usability test aims to ensure that the functionality provided by the application is user-friendly, understandable and pleasant to interact with.

3 What Am I Going to Do?

Throughout the test, you will be given a set of tasks related to the application. You are encouraged to think aloud when performing said tasks. There are no wrong answers; Try solving the task by yourself at first. If you are completely stuck, ask for advise.

Once the tasks have been completed, you will be asked to fill out a System Usability Scale, a form consisting of 10 questions with the aim of gaining an understanding of the overall usability of the system. Every question is weighed on a scale from one to five - one indicating "strongly disagree" and five indicating "strongly agree". The questions are designed in such a way that they expect you to answer them without deliberation. As such, please pick the number corresponding to your immediate reaction.

4 Tasks

- Register an account with the application. For the sake of convenience, please utilize this email address: "Y@Y.com".
- You have received an invite for the team "Trondheim beitelag". Please join it.
- Create a team named "My Team".
- Invite me to the team named "My Team". My email address is "X@X.com".
- Download a map area of Trondheim and name it "Trondheim".
- Cancel the pending download of the map area.
- Download the previously selected map. Do not cancel the download.
- Prepare for conducting an offline trip by downloading any farms containing the word "Gård".
- Remove the downloaded farm named "Min Gård".
- Start a new supervision trip within the map area named "Trondheim".
- Register a sheep observation distanced far away from your position. Specify the sheep amount as 10. Five of the sheep are gray, whereas the remaining 5 are black.
- Register a sheep observation right on top of yourself. You observe 10 sheep, all of whom are gray.
  - Specify sheep ties through the use of swiping. You observer one blue tie, two yellow ties and three red ties.
  - Add the farm "Trondheim Gård" to the sheep observation.
  - Add an unknown farm to the sheep observation. The identifying colors of the farm are red and blue.
- Register an "other" observation right on top of yourself. The observation is concerned with a dead sheep.
  - Add an image to the "other" observation. Specify that the sheep in the picture is killed by a wolf.
- View the observations for the currently ongoing trip.
- Remove the first sheep observation registered for the ongoing trip.
- Finish the ongoing trip.
- Find the previously finished trip you conducted.

- Send the finished trip to the server.
- Find the previously finished trip you conducted.
- Change your name to "Ola Nordmann".
- Delete the previously conducted trip.
- Transfer administrative privileges of your team to me.
- Leave the team you created.
- You have been granted ownership rights to the team "Trondheim Beitelag". Remove me from this group.
- Invite me back to the team "Trondheim Beitelag".
- Delete the invite you sent me to the team "Trondheim Beitelag".
- Delete the team "Trondheim Beitelag".
- Delete the downloaded map area.
- Log out of the application.

5 SUS Form

Please fill out the form. A score of one represents Strongly Disagree, whereas five represents Strongly Agree.

No.	**Question**	**1**	**2**	**3**	**4**	**5**
1	I think that I would like to use this system frequently					
2	I found the system unnecessarily complex					
3	I thought the system was easy to use					
4	I think that I would need the support of a technical person to be able to use this system					
5	I found the various functions in this system were well integrated					
6	I thought there was too much inconsistency in this system					
7	I would imagine that most people would learn to use this system very quickly					
8	I found the system very cumbersome to use					
9	I felt very confident using the system					
10	I needed to learn a lot of things before I could get going with this system					

Jonas Ege Carlsen

Digitalizing the Sheep Supervision Documentation Process

NTNU
Norwegian University of
Science and Technology