POLITÉCNICO DE LEIRIA | ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Livestock Identification and Traceability System – a case study mobile implementation based on a generic architecture for livestock management

Master's degree in Computer Engineering - Mobile Computing

João Pedro Dias da Rosa

Leiria, July of 2020

Livestock Identification and Traceability System – a case study mobile implementation based on a generic architecture for livestock management

Master's degree in Computer Engineering - Mobile Computing

João Pedro Dias da Rosa

Internship Report under the supervision of Professor Catarina Isabel Ferreira Viveiros Tavares dos Reis, PhD in the Computer Science Department of the School of Technology and Management of the Polytechnic of Leiria, and José Góis from Digidelta Software.

Leiria, July of 2020

Originality and Copyright

Acknowledgments

First, I would like to thank my company, *Digidelta Software*, for allowing me to do the internship in a project from the ground, giving me freedom to test and choose technologies and ideas.

I would like to thank Professor Catarina Reis, for accepting to be my advisor, for the continuous support, motivation, and advice along this thesis.

Last, but not least, I would like to thank my friend and colleague, José Góis, for his continuous help and interest, for showing me that any time is a good time to learn, and that knowledge is better when is shared.

Abstract

In a constantly growing market, the ability to provide quick, reliable solutions is becoming more and more important. Companies want to spend less time build the base of the project and more time in what makes it unique, perfecting it to the client's wishes. It is not surprising that a company ends up reusing the base of an old project and adapting it to new ones. As this goes on, it becomes clearer that a base project with a well-defined generic architecture would benefit the companies that do this.

Based on previous applications developed by the company, a set of requirements were defined that would need to be included in the generic architecture, such as offline mode, communication with RFID devices, and synchronization. This thesis purposes a solution, based on the recommended guidelines, of how this can be achieved, the technologies it uses, and how they interact with each other to achieve a flexible, adaptable solution.

Keywords: Android, Generic Architecture, Kotlin, Mobile

Contents

List of Figures

List of Tables

List of Abbreviations and Acronyms

API	Application Programming Interface
BLE	Bluetooth Low Energy
CRUD	Create, Read, Update and Delete
CSS	Cascading Style Sheets
DAO	Data Access Object
DLITS	Digidelta Livestock Identification and Traceability System
ESTG	School of Technology and Management
FAO	Food and Agriculture Organization of the United Nations
GPS	Global Positioning System
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
HTTP	Hypertext Transfer Protocol
ICAR	International Committee for Animal Recording
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MVVM	Model-View-ViewModel
OS	Operating System
PoC	Proof of Concept
RFID	Radio-Frequency Identification
URL	Uniform Resource Locator
UX	User Experience

1. Introduction

The complexity of creating reusable systems is something that has been acknowledged by designers and engineers for a long time. Although the process of designing something reusable is expensive, both in time and in money, is it the key speed up future developments of similar systems.

This design is even more challenging in mobile applications, as the technologies and guidelines are being built and updated every day, and an architecture can become obsolete in a not long time. As the world becomes more and more mobile, the focus in mobile development is increasing, speeding up this evolution of technologies and guidelines.

This thesis will show how a generic, native Android architecture was created, which technologies were used, their purpose, and how they interact with each other to achieve the purposed goal. A library to abstract the communication with RFID devices was also developed, which will also be used in future applications. This thesis is divided in three major areas: the introduction to the thesis, company where the internship was performed, and the internship programme; the research performed previous to the official start of the internship, where a short study of the possible technologies to use and target market was done, the technologies used and how they interact with each other, and the final result; the critical analysis, where it is reported what went well, what went wrong, what can be improved in the future, and lastly, the conclusion of the thesis and the internship.

The initial proposal was to develop a native Android application to support brigades working in the field using RFID devices. The application is called Digidelta Livestock Identification and Traceability System mobile (DLITS mobile), the mobile version of DLITS, supporting, amongst other functionalities, offline mode, interaction with RFID devices, intervention of animals, and automatic synchronization. The proposal also included the development of a generic architecture that would, not only, be used in this mobile application, but also serve as base for future similar Android projects, speeding up the development and delivery of said projects.

2. Internship Programme

The internship proposal was signed-off by *Digidelta Software* and required the development of a mobile application for Android or iOS, using, either a native or a cross-platform approach to its development. The application's name is DLITS mobile and is intended to help brigades identify animals and perform actions to them, providing offline mode, automatic synchronization, interaction with RFID devices, amongst others.

2.1. Characterization of the company

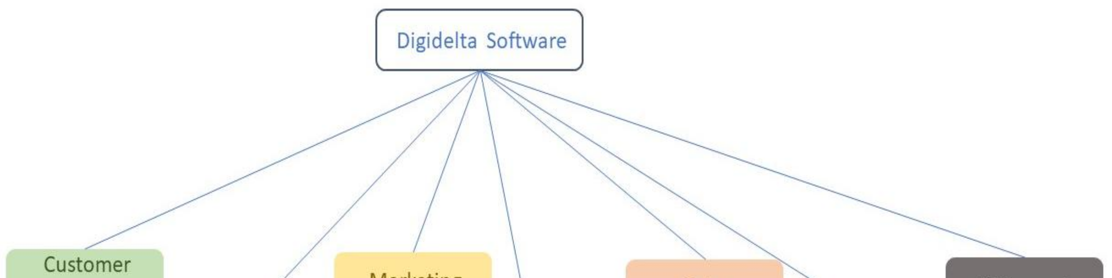
The internship took place in *Digidelta Software* [1]. *Digidelta Software* is a multi-disciplinary software development and consulting firm with long-time experience in creating tailor-made innovative solutions for various business sectors, with special focus on applications for animal health, breeders, fund management communities, and livestock management.

Digidelta Software was created in 1989, in Torres Novas. Its first major project, *Programa Informático de Saúde Animal* (PISA), was an information system capable of managing the livestock of Sociedade de Agricultores de Torres Novas. In 1990, the Ministry of Agriculture started using PISA, and after an initial test, a new version was created and eventually expanded through the entire country, making it the official livestock health system in Portugal [2].

After several name changes, the company eventually settled in Leiria under the current name, *Digidelta Software*.

Digidelta Software aims to be a worldwide reference in the development of tailor-made information system regarding livestock management. To fulfil this, *Digidelta Software* develops information systems using innovative methodologies and technologies for state and private entities in a market without borders.

Digidelta Software can be divided in seven departments, as seen in the organization chart in Figure 1.



Although each department is responsible for a specific set of tasks, they all work together and constantly intercommunicate.

- Costumer support - Responsible for interacting directly with the customers. It serves as a filter and bridge between the customers and the developers, allowing the developers to work without as many interruptions and helping the customers in smaller tasks that do not require the interaction of a developer. This department is also responsible for preliminary usability tests before allowing actual clients to test the software.
- Developers - Responsible for the entire project development process since the requirement analysis to the delivery.
- Marketing - Responsible for promoting the company's projects, finding new projects, and managing its image to the outside.
- Design - Responsible for designing mock-ups (which greatly reduce the development process time), improving the application's user interface and user experience.
- Sales - Responsible for finding customers for already developed applications (such as Wezoot).
- Financial - Responsible for handling the company's money, such as employee's payments.
- IT Support - Responsible for providing and maintaining the infrastructure that the company uses, such as the hardware (computers, keyboards, servers, internet) and the software (such as Microsoft licences), allowing each department to focus on their respective tasks.

As a developer, my main interaction was with other developers, either to help with business logic questions or opinions based on their experience on previous related projects. Since the project is not yet being tested by Customer Service or used by clients, my interactions with the Customer Service department was not as much as it would be otherwise. However, since they serve as usability testers and often know the concerns of clients, they were very important in the development in matters regarding the user's usage of the application.

Digidelta Software's clients can vary between small livestock owners and country's governments. Smaller clients usually use a common platform, shared by other small clients, while bigger clients, like country's governments have tailor-made systems to fit their specific needs. Although most of its clients are in Portugal, *Digidelta Software* has developed for other countries' official entities, such has Morocco (*SNIT.maroc*) and Botswana (*BAITS.botswana*), amongst others.

To help keep track of the livestock, *Digidelta Software*'s systems interact with a lot of hardware. Understanding and working with such hardware is critical to the vast majority of the software development done at the company. Some examples are visual and electronic earrings, RFID scanners, scales, and animal sorting sleeves. *Digidelta Software* has a partnership with *DataMars*, a livestock management and animal health company that, amongst other things, supplies earrings and RFID scanners [3]. This partnership is helpful to allow *Digidelta Software* and *DataMars* to enter international grant funding, since most of the time they involve the earrings and scanners suppliers, such as the ones presented in Figure 2. This is helpful for both sides, since *Digidelta Software* provides the software, and *DataMars* provides the hardware.

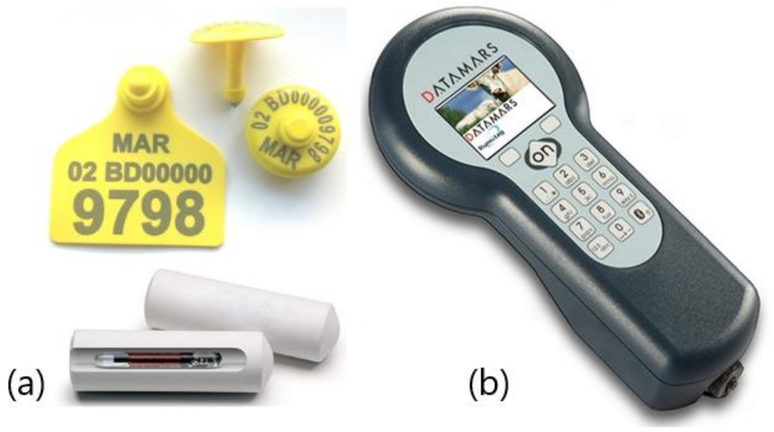


Figure 2 - a- Electronic identifiers b - RFID reader

Identified as "(a)", we can see electronic earrings (with a yellow color) that can be read with a RFID reader [4], [5]. Below the electronic earrings, we can see the ruminal bolus, a device ingested by bigger ruminants (such as cows) that stays in their stomachs. Like the electronic earrings, it can be read using a RFID reader.

Recently *Digidelta* has been involved in multiple new projects. One of those projects is a system to digitalize the livestock markets, collecting business data to later inform of strategic decisions for trade and markets. It was tailor-made to *Amfratech* and named (by the client) *Mifugo Data* (which means Livestock Data in Swahili) [6]. A WebApp and a mobile application were both developed. *Digidelta Software* has also been developing a tailor-made system to help manage maternities to produce clam larvae called *Oceano Fresco* (the final name is not yet assigned). The software will interact with sensors capturing temperature, salinity, phytoplankton levels, amongst others, and alert the staff to adjust the levels, or adjust them automatically if possible.

Digidelta Software is currently a medium sized company. Everyday a world of opportunities rises, and the industry market is becoming more demanding than ever with new businesses appearing. To fulfil the ever-growing market requests, the company is growing with it.

2.2. Methodology

When it comes to requirements. the user must be able to list the holdings associated with the previously selected entity, register new holdings, and update existing ones. A holding is a local (such as a farm) that can have multiple owners (the people who own the holding), keepers (the people who take care of the holding and its animals) and animals, following the FAO and ICAR guidelines [7], [8]. The user must be able to check a given holding's owners, keepers, and animals. When it comes to owners and keepers, the user must be able to list the ones related with a given holding, create new ones and edit existing ones. The user must be able to list the animals present in each holding, create new ones, and edit existing ones.

As for services, the user must be able to register multiple actions to multiple animals at the same time, identifying the animals either manually or with RFID readers. The user must be able to consult previously made services and edit them if they are yet to be finished.

The user should also be able to consult the summaries of the current session, that includes, amongst other things, the services made.

The application must be able to work offline, saving the operations performed, except the login and entering an entity, which must be performed online. The application should regularly upload the user's data to the remote API, so it becomes available to other users. Data relevant to the user should also be regularly updated so he uses up-to-date data.

The application should be able to handle a high amount of data and users, while still performing well. The application should also be intuitive, allowing users to use it without much troubles. Lastly, the application should be easy to modify and adapt to different scenarios.

The application's goal is to be used both for online and offline scenarios. Amongst other things, the proposal required skills in Kotlin and Android, Swift and iOS, mobile cross-platform frameworks, and Git, as the source code versioning management system.

One of the purposes of the internship was to develop a generic architecture to be adapted to future projects, speeding up the applications' development process in the future. This necessity comes from having multiple applications with the same common requirements, difficulties, and functionalities, such has synchronization and integration with external devices. Finding a way to standardize these processes and use them in future applications will bring value to the company, far beyond the application's value by itself.

In the beginning of the internship, a preliminary version of the work plan was made. During the period of the internship and evolution of the work, the plan suffered minor adjustments reflected in the Gantt Chart presented in Figure 3.

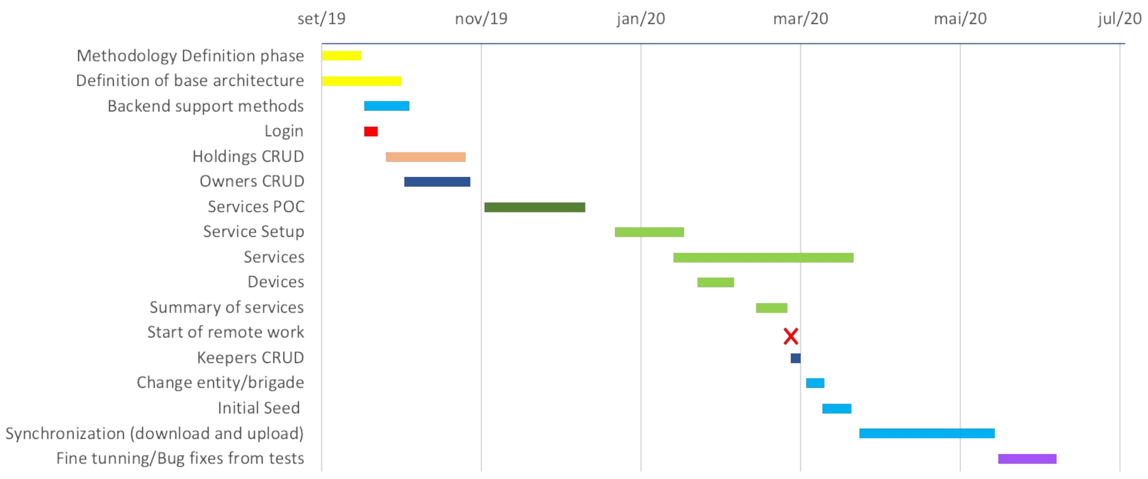


Figure 3 - Gantt Diagram

The first month of the internship was dedicated to the definition of the methodology where responsibilities were assigned, the Scrum artifacts and events were defined (Sprints, and conflicts. Therefore, this phase affected the whole project and allowed a much smoother transition into later phases.

The next month of the internship was the phase of developing the core architecture. Multiple versions of the architectures were tested until the team accepted one. The goal of the architecture was to make the application reactive, rather than proactive. This phase was crucial since the entire development of the application and all its processes are grounded on the generic architecture. A change in the architecture could require a great amount of time and effort since it involves the entire application. The development of a generic architecture to be used in multiple projects was the goal, and accepting an architecture known to be flawed would mean that either its flaws would be replicated through multiple applications or each new application would need to fix its flaws, failing the task of creating a generic architecture. Therefore, it was necessary to test multiple architectures until one was discovered that could fit both the application's needs as well as envisioning the requirements for future applications. Although we knew the architecture would have to receive small improvements as the application grew and new challenges appeared, we were confident that no major changes would be required.

The next phase was the development of the application's core functionality, the launch of services. Several minor improvements were made to the architecture, to better accommodate the requirements. First, a proof of concept (PoC) was developed; to test the idea we had designed for the functionality was feasible and accurate. The PoC included the setup of the service and the actual service along with a device's module integration.

Around the beginning of March, due to the pandemic outbreak, the company started working remotely, maintaining the normal work schedules. This created a few setbacks. The Devices Module development stopped since it needed specific devices (such as the RFID readers) to test and that were left in the company, which was isolated for two months. In addition, the lack of presential communication difficulted the usability tests by the users (other company employees).

After that, the focus was automatic synchronization. Synchronization was necessary to make sure the work done on the application is integrated with the remote API, and that the application is using the most recent data present, in the remote API.

The final step was the publish of the application to the Google Play, where a beta version was published and given access to a closed group of testers. The testers were given a questionnaire, where they could rate how hard it was to perform certain tasks. An area to leave comments on each section was also provided, allowing the testers to explain why they had difficulties doing something or leave suggestions of improvements. The questionnaire can be consulted in Appendix D – Usability Tests Questionnaire.

2.3. Preliminary study – Technology Selection

Previously to the start of the internship, it had to be decided which technologies would be used in the application. As stated before, there were several options available: native (for Android and iOS) or cross-platform (using a hybrid development framework that served those two operating systems by providing both with specific binaries generated from the same source code). Since I was already a member of the company before the internship, I was able to take part on this phase, helping in the decision of the technologies. The result of this preliminary study is presented in a resumed way in chapter 3 – Related Work and can be consulted with more details in Appendix A – Technology Selected and Appendix B – Kotlin vs Java.

To help with the decision process, four versions of the same demo application were developed: a native version for Android with Kotlin, an hybrid version developed with Ionic, an hybrid version developed with NativeScript and lastly, an hybrid version developed with React Native. To assure that a good comparison was made, the demo application provided the same functionalities that were implemented in all versions: a master-detail view of animals and interaction with Bluetooth devices.

Considering the team's previous experience with Android (using Java) and with Angular (Ionic and NativeScript), React Native was the less-known technology by the developers. After a few tests, it was decided to exclude it since it would bring a greater learning curve than what it was intended. After that, the other three versions were presented to the company. The team presented them without sharing the information of which technologies were behind each version, to prevent biased opinions. Both the aspect and usability of the versions were considered, and the selection led to the one that had a more natural feeling to its usage.

After the study and presentations, the team decided that native Android with Kotlin would be used. The study can be read in further detail in appendix X.

3. Related Work

Mobile app development is the process of developing apps for mobile devices, such as smartphones and smartwatches. With the increase in numbers of mobile devices per user, there are more and more mobile devices which can support mobile applications. The need for said applications grows and, with it, the need for standards to help develop them.

Looking at similar applications can give a good idea of how they work, what they did right and what can be improved. We started looking at some of the company's applications, and then to some external ones.

- Pisa Mobile
  Despite being developed for laptops in Windows Forms, and not exactly smartphones, it is considered a mobile application as it has the functionalities that this kind of mobile applications have: offline capabilities and connection to external devices, such has stick readers and printers [9]. **It is used for applying actions animals in the field. DLITS services are based on this functionality** . This version is used usually in larger holdings, where its more practical to sit and register the interventions with multiple people working on it.
  It has a simpler interface compared to its desktop version (Pisa.NET). It works with a manual synchronization mechanism. It was developed by *Digidelta Software*.
- Pisa Pad [2]
  Pisa Pad, developed by *Digidelta Software* , is the Android version of the current Pisa, Pisa.NET. Its and Android application with offline functionalities. To synchronize its data, Pisa Pad uses a manual file system, importing a file to seed the local database before starting to work, and exporting it at the end of the work. Like Pisa Mobile, it can connect with RFID stick readers and portable tag prints to place on test tubes with blood samples, via whether Bluetooth or Wi-Fi. It has the same purpose as Pisa Mobile, allowing the users to choose which platform they prefer to work on, choosing the version that better suits them.
- Wezoot [10]
  Wezoot, developed by *Digidelta Software* , is an Android application designed for livestock producers, to help ease their work and increase productivity. It supports connection with external devices, such as draft gates and scales, which was one of

the main motivations for developing the Devices Library. Unlike Pisa Mobile and Pisa Pad, **Wezoot's synchronization system is automatic. DLITS synchronization system is based on this functionality**.

Amongst other things, Wezoot offers an alert system paired with a task manager to help producers stay on time.

- Yagro

  Yagro is an input management ordering tool that connects farms directly to their suppliers, helping farmers manage their input costs and boosting their productivity [11]. It has as main functionalities:
  - Receiving quotes directly from UK's leading suppliers
  - Real-time visibility of the costs
  - Online marketplace for farm products
- Herdwatch

  Herdwatch is a livestock recording and management app that registers animal births, manages breeding-cycle information, stores animal movements and holds individual health records [12]. Amongst other functionalities, it has:
  - Automatic synchronization
  - Paperless movements and permits
  - Fast calf registration
  - Weight recording
- Farmwizard Beef Manager

  Farmwizard is an app that enables farmers to record breeding data, herd health, cattle movements and field records [13]. It connects with supported Bluetooth scales to retrieve the animal's weight. It has offline mode and synchronizes the data via internet when available.

In Table 1 we have a comparison between the relevant features of the applications.

Table 1 - Application comparison

	Application/Feature	DLITS
	Pisa Mobile	
✓	Intervention of animals	
✓	Offline mode	
✓	Connection to external devices	
	Pisa Pad	
✗	Manual synchronization	
	Wezoot	
✓	Automatic synchronization	
	Yagro	
✗	Live updates of data	
	Herdwatch	
✗	Paperless movements	
✓	Fast animal registration	
	Farmwizard Beef Manager	
✓	Digitalization of processes	

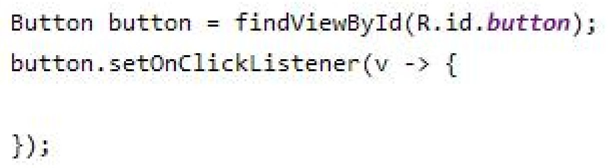
To help with the decision process on the technology to use, four versions of the same demo application were developed: a native version for Android with Kotlin, a hybrid version developed with Ionic, an hybrid version developed with NativeScript and, lastly, an hybrid version developed with React Native. To assure that a good comparison was made, the demo application provided the same functionalities that were implemented in all versions: namely, a master-detail view of animals and a connection to Bluetooth devices.

Considering the team's previous experience with Android (using Java) and with Angular (Ionic and NativeScript), React Native was the less-known technology by the developers. After a few tests, it was decided to exclude it since it would bring a greater learning curve than what it was intended. After that, the other three versions were presented to the company. The team presented them without sharing the information of which technologies were behind each version, to prevent biased opinions. Both the aspect and usability of the versions were considered, and the selection led to the one that had a more natural feeling to its usage: **native Android.** A more detailed study can be read in Appendix A – Technology Selection.

After the presentations, it was needed to decide between using Java or Kotlin as the development language.

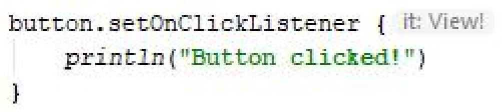
A study was made to decide which language, Java or Kotlin, would be a better fit. Amongst other things, some of the factors evaluated were the functionalities, the conciseness of the syntax and the amount of boiler-plate code necessary.

In Figure 4 we see the code necessary to catch a click event in a button using Java, and in Figure 5 the same functionality in Kotlin. Although the difference might not seem much, it's a good example of the conciseness of Kotlin, which was a deciding factor.



```
Button button = findViewById(R.id.button);
button.setOnClickListener(v -> {

});
```

Figure 4 - Catching a click event in Java



```
button.setOnClickListener { it: View!
    println("Button clicked!")
}
```

Figure 5 - Catching a click event in Kotlin

This study can be read in further detail in Appendix B – Kotlin vs Java.

After the studies and presentations, the team decided that **native Android with Kotlin** would be used.

After choosing the technology to be used and analyzing the previous applications developed by *Digidelta Software*, what worked for them and what we know that could be improved, both in usability and in their core technologies and architecture, it was decided the application would follow a generic architecture, **using automatic synchronization, and an external devices library being developed as a separate module**. This way not only would the application support and improve the development of future applications, but also allow room for customization on itself, being able to adapt with relative ease to changes new customers might need.

4. Core Architecture

This chapter will present different components of the architecture developed and how they work together to fulfill the application's requirements as well as explaining how the architecture can be used for future applications with similar requirements.

This application follows Model-View-ViewModel (MVVM), a software architecture pattern recommended by Android. One of MVVM's benefits is the separation of the development of the view from the development of the business logic. Between the views and the model, there is the ViewModel, which is responsible for exposing the data from the model in a way that is easy to present by the view. The ViewModel can control most of the view's display logic and acts as a mediator, organizing the access to the backend.

In Android, the Activities, Fragments, and their layouts represent the View. The ViewModel are the ViewModels, which merge data from multiple places to expose it properly formatted to the view. The Model is made by the services and repositories, which enforce the business logic and handle data.

In Figure 6 we have a simplified representation of MVVM, based on Android's architectural common principals [14].

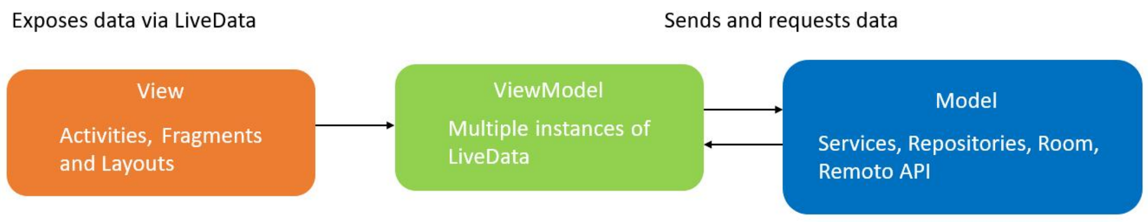

Exposes data via LiveData
Sends and requests data
View
Activities, Fragments and Layouts
ViewModel
Multiple instances of LiveData
Model
Services, Repositories, Room, Remoto API

Figure 6 - MVVM

In the application's architecture, each layer has a specific function, following the separation of concerns: The Model Layer retrieves and formats the data properly; the ViewModel Layer merges and exposes multiples data sources to satisfy the view's needs; and the View Layer displays the data to the user. Each layer has its responsibilities, and changes in one layer may not need to be applied to other layers (depending on the changes).

Based on the best practices and the guidelines reported by the Android development community, an architecture diagram was drawn, with the layers and their respective technologies, shown in Figure 7.

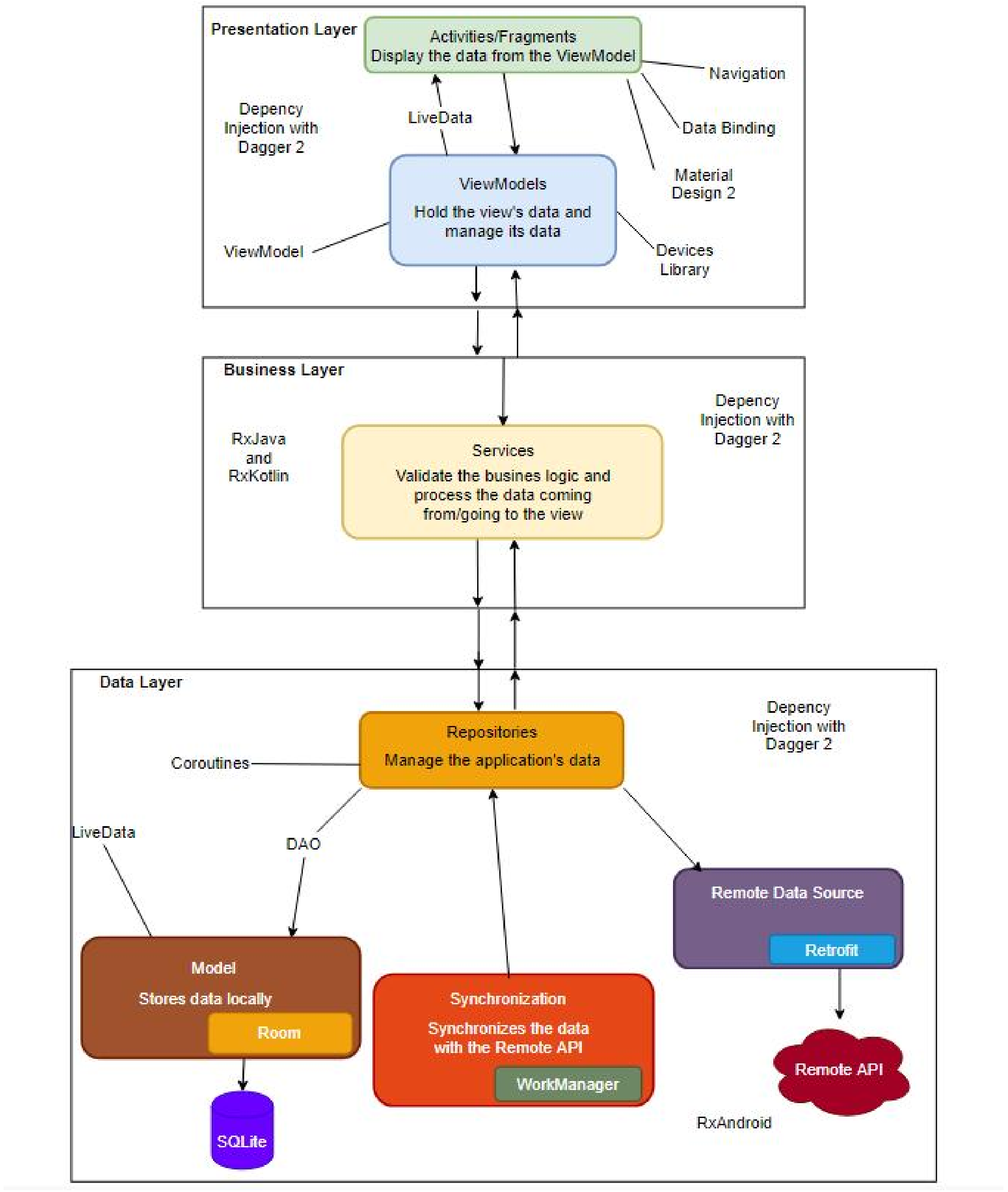

Presentation Layer
Activities/Fragments
Display the data from the ViewModel
Navigation
Depency Injection with Dagger 2
LiveData
Data Binding
Material Design 2
ViewModels
Hold the view's data and manage its data
ViewModel
Devices Library
Business Layer
Depency Injection with Dagger 2
RxJava and RxKotlin
Services
Validate the busines logic and process the data coming from/going to the view
Data Layer
Depency Injection with Dagger 2
Repositories
Manage the application's data
Coroutines
LiveData
DAO
Remote Data Source
Retrofit
Model
Stores data locally
Room
Synchronization
Synchronizes the data with the Remote API
WorkManager
Remote API
RxAndroid
SQLite

Figure 7 - Architecture diagram

4.1.Presentation Layer

The views and the respective ViewModels that manage their data compose the Presentation Layer. A view is an Activity or a Fragment that is responsible for the interface in which the user will interact with the application; it is therefore known as the top layer of an application [15], [16].

According to the separation of concerns, no business logic should be implemented in an Activity of Fragment; it should be implemented in the view's ViewModel. The view should listen to changes in the ViewModel and react accordingly.

Android suggests Single Activity Application, which is an app that has an activity to serve as an entry-point and uses fragments to create the entire GUI. However, to better organize the flow of the application, it was decided that the app would consist of multiple activities, one per area. Once in a certain area, for example, Animals, fragments are used to display, create, and edit the data.

Initially, each activity would have a ViewModel that would hold all the data shared by the activity's fragments. That ViewModel would store and manage the GUI-related data. The ViewModel is also used to communicate between fragments. However, this approach would prove to be unpractical, since there would be a repetition of operation every time a fragment was used in a different place (since it was the activity's ViewModel that would implement the logic). This would cause some ViewModels, especially those of activities with multiple fragments, to be overwhelming and complex. This code repetition and complexity would prevent scalability, since no one would be able to improve the code, and the addition of a single fragment could bring a lot of work.

Ideally, each fragment would have its own ViewModel to hold and manage its data, preventing the repetition of operations every time the fragment was used in a different activity. For example, a fragment to list animals would fetch its animals and display them. However, this would prevent the fragment from being reused, since we might want to display different animals in different contexts. For example, we might want to display all the animals of an entity in one place, and only the animals of a certain holding in other place. This would mean that a different fragment was necessary every time the source of the data was different, which would cause a big repetition of code and reduce standardization (since we want to list animals always the same way, with the same behaviors).

To fix these problems, a third approach was made, a merge between the first, which allows the reusability of fragments, and the second, that give fragments independence and greatly reduces code repetition. This approach consists of each fragment having two ViewModel. One ViewModel (the ViewModel of the activity) is used to make the communication fragments-activity and serves as the control point for all the fragments in it. The other ViewModel implements that specific fragment's logic, its behaviors, etc. For example, the fragment that lists animals has a search functionality, which is implemented in the ViewModel of the fragment. When an animal is clicked, the fragment communicates it to the ViewModel of the activity, which, depending on the activity, might signal the activity to navigate to that animal's details. On other activity, something else might be done when an animal is clicked. The important part is that the fragments do not perform outside actions (such has navigation to other fragments), allowing them to be reused.

Now, we will explain the technologies being used in this layer, and how they interact with each other to create the GUI.

LiveData

LiveData is a lifecycle aware, event-based, stream of data, which means it respects the lifecycle of the components listening, such as activities, fragments, and other lifecycle owners [17].

Besides preventing memory leaks and handling of components lifecycle, LiveData allows to have constantly updated data across the entire application (especially when used with Room) and react to configuration changes (when used with ViewModel), such has the rotation of the screen, making it fundamental for any application that handles lots of data.

ViewModel

ViewModel is a class responsible for preparing, holding, and managing data for one or multiple activities or fragments. In this case, it also handles the communication between the respective component and the business logic classes (such as the services) [18]. The ViewModel will be alive until the view (activity or fragment) is destroyed. Contrary to the view, the ViewModel will not be destroyed for configuration changes (e.g. the screen is rotated), maintaining its data intact. When the view is recreated, it connects to the same instance of the ViewModel, receiving the data that was stored in it. This works particularly well with Room and LiveData.

ViewModels can hold LiveData that is listened by one, or multiple views, allowing them to communicate between each other.

To allow the reusability of fragments, the fragment data source is defined ty the controlling activity, so that the same fragment can display different data, according to the context.

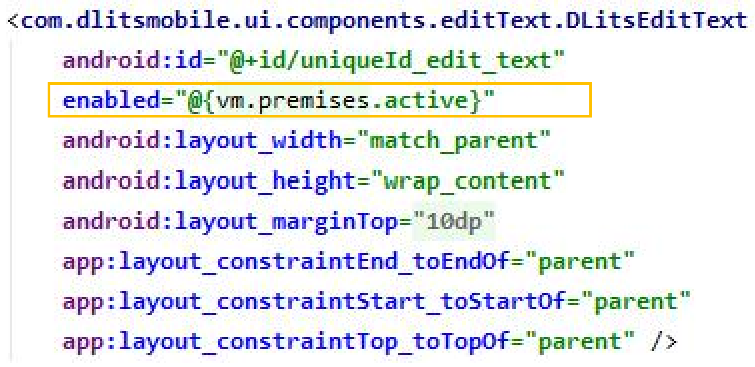
Fragments are unaware of anything outside their ViewModel, therefore certain operation, such as navigation between fragments, must be implemented by the activity when reacting to the activity's ViewModel.

Fragments also have another, internal ViewModel, that makes operations the fragments need to work properly, preventing the repetition of the same code everywhere the fragment is used. For example, in the fragment that holds a holding's details, the generic ViewModel passes the holding's id to the fragment, and the fragment's specific ViewModel will take that id and fetch all the data necessary to show/edit the holding.

Data Binding

Data Binding is a library that allows the binding of GUI components in the layout to data sources [19]. This will cause the GUI components to update when the data source updates. It is possible to bind variables, events (such has clicks), amongst other things.

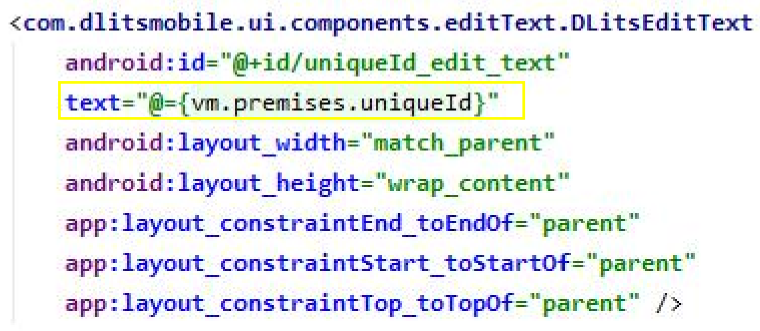
Shown in Figure 8, we can see that the component's enabled state is bonded to the object's (called "premises") active state, using one-way data binding ("represented by the "=" before the "@"). The component's state will change when the object's "active" property changes. This object is present in the ViewModel, which is not necessary, but recommended since it is the ViewModel that should hold the view's data.



```
<com.dlitsmobile.ui.components.editText.DLitsEditText
    android:id="@+id/uniqueId_edit_text"
    enabled="@{vm.premises.active}"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="10dp"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Figure 8 - One-way data binding

It also supports two-way data binding, which means that not only the GUI updates as the data source changes, but the source changes when the GUI changes, allowing to keep the GUI and the data source synchronized [20].



```
<com.dlitsmobile.ui.components.editText.DLitsEditText
    android:id="@+id/uniqueId_edit_text"
    text="@={vm.premises.uniqueId}"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Figure 9 - Two-way data binding

In Figure 9 we can see a custom component to encapsulate a specific behaviour, in this case, an edit text with extra behaviours. Although custom components do not come with data binding implemented by default, it is possible to implement it and customize it, allowing great flexibility.

Material Design 2

Material Design is a set of guidelines that illustrate the classic principals of good design[21]. They tell how an application's components should interact, in either a broad, or a detailed perspective. Amongst other things, the guidelines talk about colors, which components to use and how they should react, motions, transitions, icons, etc. Material Design 2 is a direct upgrade of Material Design 1, both developed by Google. They are used not only to make good looking applications, but also intuitive. Although they are not specific for Android, there is a specification for it [22].

There is a great need to make the application as simple and clean as possible, allowing its users to figure out what they need to do and how to do it with as few clicks as possible.

Considering most users might not be tech-fluent, a complicated interface will cause them to easily give up on doing what they were trying to achieve. Following the Material Design 2 guidelines means shortening a lot of user-acceptance work, since we did not need to test some of the design choices as they were previously tested and therefore recommended by Google. Although there is no solution that fits every user, we can assume a good majority will be satisfied.

Navigation

Navigation is an Android library that facilitates handling the navigation between different pieces of content in the application [23]. It can help implement navigation from simple events, such as a button click, or more complex events, such as app bars and navigation drawers. It also assures a consistent and predictable user experience, according to Android's navigation principles [24].

With Navigation we can create navigation graphs (called navGraphs) which define the interaction between fragments (or activities) and their animations. We can also have nested navGraphs inside another navGraph, to better isolate certain processes (for example, the master-details view of animals).

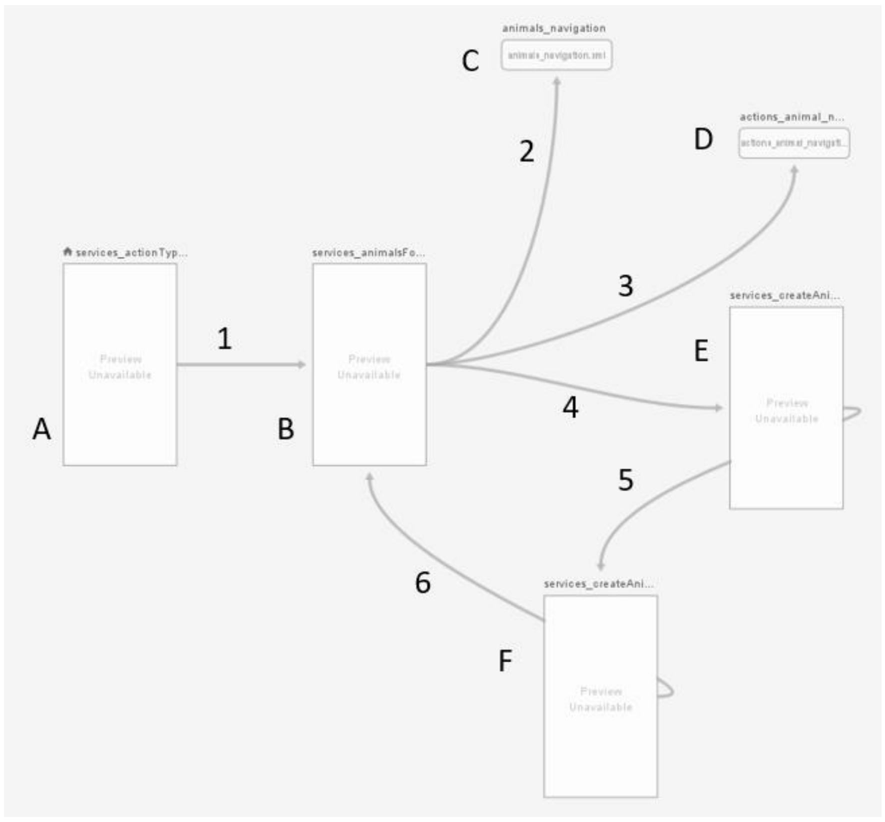


Figure 10 - Navigation example

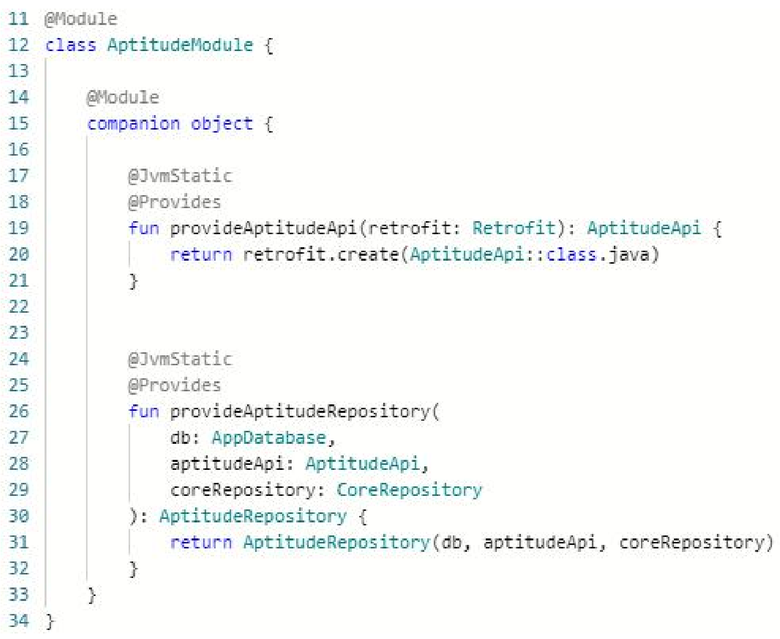
In Figure 10 we have a navigation graph. Each letter is a fragment, except "C" and "D" which are nested navigation graphs, that represent master-detail views. Each number represents an action: number “1” is the navigation between fragment “A” and fragment “B”, and so on. Each navigation (between fragments) can have different animations, but for consistency, similar navigations should have similar animations.

Dependency Injection with Dagger 2

Dagger is a compile-time dependency injection framework for both Java and Android [25]. The earlier version was developed by Square and is now maintained by Google. Dagger is a replacement for the common Factory classes that implement dependency injection design patterns, without the need to write all the boilerplate code. It helps the developer focus on how the classes work, rather than how to get their dependencies satisfied.

Using Dagger, we can create simple modules that provide a certain area (for example, Aptitudes, shown below) and inject it wherever it is necessary. New instances of the injected classes are created every time they are injected somewhere, which means that two places using the same modules will have different instances of the same classes. Because of this, it is important that those classes are stateless, which means they should not have a state, rather just perform operations. However, it is possible to mark methods of a module as Singletons, meaning that always the same instances are injected.

In Figure 11 we can see a Dagger2 module, in this case, AptitudeModule, which provides the AptitudeApi and AptitudeRepository.



```
@Module
class AptitudeModule {

    @Module
    companion object {

        @JvmStatic
        @Provides
        fun provideAptitudeApi(retrofit: Retrofit): AptitudeApi {
            return retrofit.create(AptitudeApi::class.java)
        }


        @JvmStatic
        @Provides
        fun provideAptitudeRepository(
            db: AppDatabase,
            aptitudeApi: AptitudeApi,
            coreRepository: CoreRepository
        ): AptitudeRepository {
            return AptitudeRepository(db, aptitudeApi, coreRepository)
        }
    }
}
```

Figure 11 - A Dagger2 module

Devices Library

Devices Library is an internal library being developed to abstract the communication with external devices, whether via Bluetooth, BLE or Wi-Fi [26], [27]. The purpose of this library is, not only support this application, but also support other company's applications, standardizing and abstracting the way the applications interact with devices. This library can be used outside the application, since it was developed as a separate module, that can be integrated in other applications.

The devices the application will communicate with can vary from small stick readers to large draft gates. The library is be able to detect nearby devices and connect to them, abstracting from the application the way the connection and communication is made (regardless of the device and its communication type).

Depending on its usage by the user (and if the device allows it), the readings from the devices can either be done one by one (a reading is performed and emitted) or in batch, called sessions (multiple readings are done and emitted all at once). The library must handle both scenarios.

Some devices have non-standard data protocols, which means that they send data in a non-standard format. To deal with this, a generic implementation is made (that handles how most devices work), and a specific implementation is made for that specific devices. The API provided must be the same, regardless of the devices since the purpose of the library is the abstraction of the procedures.

In chapter 5 - DLITS, we will see interactions of the application with this library.

Together, ViewModel, LiveData, Data Binding, Material Design 2, Navigation, Dagger 2 and the Devices Library create the GUI. ViewModel holds and manages multiple instances the view's LiveData, presenting it in an updated, easy-to-read manner. Using data binding, we can bind the LiveData directly from the ViewModel to the GUI, using either one-way or two-way data binding, according to the necessity. With Material Design 2, we can create a clean, easy to use GUI for the user, promoting better usability. With Navigation, we can navigate between views, such as fragments, with standardized animations, in an easy to implement way. With Dagger2, we can make sure every ViewModel has its dependencies fulfilled, allowing us to focus on the implementation.

4.2. Business Layer

The Business Layer acts as a middleman, between what the user does (in the Presentation Layer), and what is saved to be synchronized (in the Data Layer), enforcing the business logic. This layer is composed by multiple services that validate the requests they receive before performing a certain operation.

Each service is responsible for managing a section of the data. For example, the AnimalService is responsible for managing requests related to animals, such as creates, updates, reads, amongst others. A service might have access to multiple services, to fetch data from different areas to validate the requests performed (for example, the creation of an animal). If the request is valid, it forwards it to the respective repository (in this case, AnimalRepository), that will create the animal, not validating it, since it is already validated.

This layer allows for standardization of the processes; since the same operation can be performed in multiple contexts, we want to make sure it is processed the same way every time. ViewModels do not have direct access to repositories, only via services, so we can guarantee that when the implementation of an operation (like a create) changes, only the service needs to change, and the ViewModels remain the same.

This layer is also responsible for handling the user’s permission. Each user has a role associated with him in each entity he is associated with. Depending on the user’s permissions, the Presentation Layer might disable/hide some options and functionalities.

In this layer, the technologies used are RxJava, RxKotlin and Coroutines.

RxJava and RxKotlin

RxJava and RxKotlin are part of Reactive Extensions (shortened to ReactiveX), a library for composing asynchronous and event-based applications. ReactiveX operates on discrete values that change over time [28]. Instead of getting the current state of an object, it observes its state, and allows the application to react to it. By using Observables, we can treat streams of asynchronous events with simple, easy-to-read, and less error-prone code.

RxJava is Java's implementation of Reactive Extensions [29]. RxKotlin is a lightweight library that adds convenience functions to RxJava. Although RxJava can be used out-of-the-box with Kotlin, RxKotlin makes the usage of those feature more *Kotlin-like* [30].

They are used to sometimes communicate with the Data Layer, such as in data validations, that are not directly GUI-related. However, most of the communication between the Business Layer and the Data Layer is made via LiveData.

Like in Presentation Layer, Dagger2 is used to fulfil dependencies, in this case, the service's dependencies.

Together, RxJava, RxKotlin, and the application of the business logic create the Business Layer. This layer is does not rely very heavily on technologies, since its work is business validation. Most of it are requests to the Data Layer to make sure the requests coming from the Presentation Layer are valid. After a request is deemed valid, it is processed and sent to the Data Layer.

4.3. Data Layer

The Data Layer is made by the repositories (each responsible for handling data of a certain area), the local database (supported by Room), the synchronization (with Work Manager) and the communication with the remote API (with Retrofit, integrated in each respective repository).

This layer responsible for managing the application's data. It has no business logic implemented; its only concern is making sure the data is valid. This layer is responsible for, amongst other things, serving as an entry point to the remote API and Room. It converts the objects coming from the remote API to application's models to be inserted in Room and merges Room requests to facilitate its exposure to the other layer.

When the Business Layer makes a request to create, for example an animal, it provides a model used to display it in the Presentation Layer, which might not be the same model as the one saved in Room. After the Business Layer validated the animal, it requests its creation to the Data Layer, which assumes its already valid, and won't validate it again. It will transform that object into the different appropriate models, and insert them into the necessary tables, creating/updating the relations, if any.

The Data Layer is also responsible for keeping track of the data that is yet to be synchronized. Each object that can be created/update in the application (such has Holdings, Animals, Owners, Keepers, amongst others that the user doesn't create/update directly, such as relations), has two flags that annotate the object as either new ("isNew") or updated ("isUpdated"). Any object that has either of these flags set to "true" needs to be synchronized. After its synchronization, the objects flags are set to "false".

The technologies used in this layer are Room, DAO, Coroutines, Retrofit, Work Manager, RxAndroid, LiveData and Dagger2.

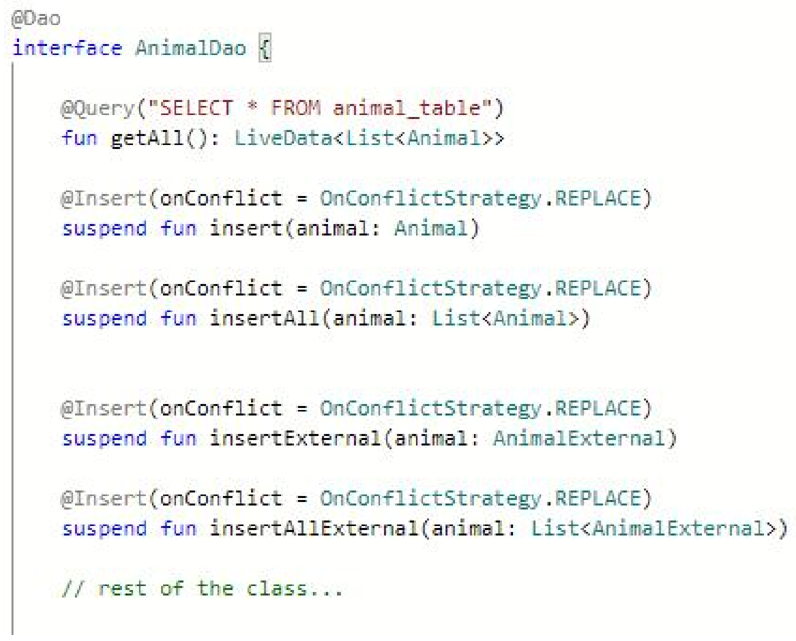
Room

Room is a persistence library from Android's Architecture Components that provides an abstraction layer over SQLite to allow for a more robust database access [31]. It has all the functionalities of SQLite, being a relational database with Primary Keys, Foreign Keys, and relation validations. Room's models were generated via T4 Text Template [32].

Room serves as a cache database, allowing for the application to work while offline and is used as the single source of truth, working very well with LiveData. Single source of truth means that all data used in the application comes from Room, never directly from other places, such as the remote API. Even though some repository's methods are responsible from fetching data from the remote API, that data will be put into Room, and not passed to the application directly. Only repositories interact directly to Room via DAOs (Data Access Object) [33].

DAO

DAOs are the classes where we can define the interactions with the database, in this case, Room. A DAO is an interface marked as DAO with the @Dao annotation. Each method has an SQLite query annotated above, representing the query it will run against Room. Also, we must specify the return type of the method. Although the type of the object must match the SQL query (for example, if we say the method returns an Animal, the query must return an Animal), we can specify if it comes as LiveData, regular objects, as we can see in Figure 10.



```
@Dao
interface AnimalDao {

    @Query("SELECT * FROM animal_table")
    fun getAll(): LiveData<List<Animal>>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(animal: Animal)

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertAll(animal: List<Animal>)


    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertExternal(animal: AnimalExternal)

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertAllExternal(animal: List<AnimalExternal>)

    // rest of the class...
```

Figure 12 - Example of a DAO

Coroutines

Coroutines are light-weight threads [34]. Coroutines help make asynchronous requests feel more fluid, and avoid multiple call-backs inside each other, which could create a normally called "call-back hell". Coroutines give a more synchronous feel to asynchronous programming, making the application easier to develop and maintain.

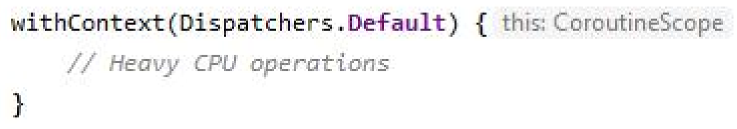
Since LiveData with Room was used to fetch the data, there is no need to use coroutine to access Room (which should only be access asynchronously) for normal GET methods. However, when validating the data (for example, to create/update an animal), we need to implement the business logic against the database. In these cases, coroutines were used to make simples GETs, fetching as little data as possible, since making an asynchronous request synchronous still takes time.

Coroutines can use dispatchers [35]. A CoroutineDispatcher determines what thread or threads the coroutine uses for its execution. It can confine the coroutine to that thread, assign it multiple threads or let it run unconfined.

There are four dispatchers: Unconfined, IO, Default and Main.

- Unconfined: The coroutines uses whatever thread the calling function is using, without confining to any specific thread.
- IO: This dispatcher is optimized to perform disk or network I/O outside of the main thread. Examples of good usage of this dispatcher include using Room, reading from/writing to files, and performing network operations.
- Default: This dispatcher is optimized to perform CPU-intense operations outside the main thread. Examples of good usage of this dispatcher include sorting lists and serializing/deserializing data.
- Main: This dispatcher should be used to perform operations that interact with the GUI. and, therefore, should be performed on the main thread. This dispatcher should only preform quick work, since it could block the main thread, blocking the GUI.

Shown in Figure 13 is a coroutine with dispatcher Default.



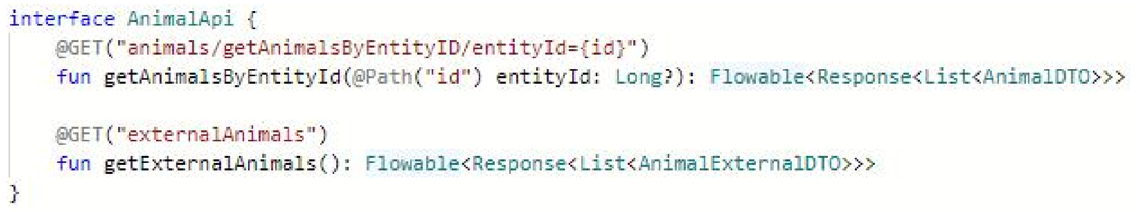
```
withContext(Dispatchers.Default) { this: CoroutineScope
    // Heavy CPU operations
}
```

Figure 13 - Coroutine example

Retrofit

Retrofit is a type-safe REST client for Android and Java by Square [36][37]. This library provides a framework for interacting with remote APIs and making network requests with OkHttp (an efficient HTTP and HTTP/2 client for Android and Java applications) [38][39]. Retrofit also facilitates the deserialization of JavaScript Object Notation (JSON) objects to Plain Old Java Objects (POJO), which must be defined for each "resource" in the response.

A Retrofit service is an Interface with methods annotated as Http methods (can be @GET, @PUT, @POST, etc…), and the respective paths, as seen in Figure 14.



```
interface AnimalApi {
    @GET("animals/getAnimalsByEntityID/entityId={id}")
    fun getAnimalsByEntityId(@Path("id") entityId: Long?): Flowable<Response<List<AnimalDTO>>>

    @GET("externalAnimals")
    fun getExternalAnimals(): Flowable<Response<List<AnimalExternalDTO>>>
}
```

Figure 14 - AnimalApi example

Since we only need one instance of Retrofit, we can declare it in the AppModule to make it a Singleton. To create a Retrofit instance, we can use its builder, give it the base URL of the remote API it is consuming and, the converter factory, which is the factory used to deserialize the responses from JSON to POJO. In this case we are using Gson, Google's parser [40].

Work Manager

Work Manager's purpose is to reliably schedule deferrable, asynchronous tasks that are supposed to run even if the app exits or the device restarts [41]. Amongst its functionalities, Work Manager allows to:

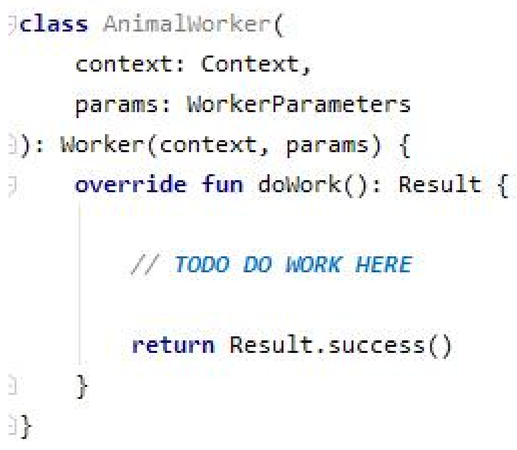
- add constraints to requests (for example, available internet connection, the device being idle)
- chain tasks (one task running after other successfully finishes)
- schedule repetitive tasks to be ran after a certain period

Work Manager is not supposed to be used if the purpose is to run the tasks on an exact moment; its purpose is to make sure a task runs, even tasks schedule to run immediately might take a while to start, since it's the operating system that handles them. There is no way to force a task to run when we want, only in a certain time frame.

All these functionalities make it ideal for synchronizing the data with the remote API. To make API requests, we need to make sure we have internet connection. We can even constraint the connection to make sure it only syncs the data using an unlimited data connection, to avoid performing such a big data transfer using limited data plans. We can also constraint the requests to not run when the device has low battery, to prevent its unexpected shutdown (user see device has enough battery, synchronization runs without user being aware and drains the battery).

Work Manager is used to both download the data from the remote API and to upload the data changed to the remote API.

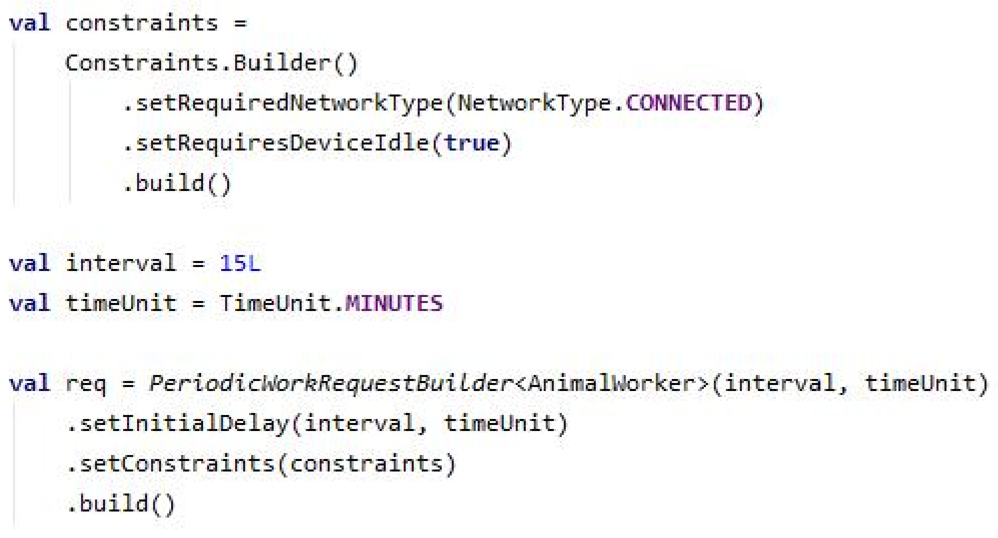
To create a Work Manager, simply create a class that extends Worker, then implement "doWork()", as seen in Figure 15. If we want our worker to use coroutines, extend CoroutineWorker instead.



```
class AnimalWorker(
    context: Context,
    params: WorkerParameters
): Worker(context, params) {
    override fun doWork(): Result {

        // TODO DO WORK HERE

        return Result.success()
    }
}
```

Figure 15 - AnimalWorker dummy implementation

After creating the worker, we need to create a request that uses it, shown in Figure 16.



```
val constraints =
    Constraints.Builder()
        .setRequiredNetworkType(NetworkType.CONNECTED)
        .setRequiresDeviceIdle(true)
        .build()

val interval = 15L
val timeUnit = TimeUnit.MINUTES

val req = PeriodicWorkRequestBuilder<AnimalWorker>(interval, timeUnit)
    .setInitialDelay(interval, timeUnit)
    .setConstraints(constraints)
    .build()
```

Lastly, we enqueued the request. A request can have multiple workers, and a worker can be used on any type of request, since it is unaware of the request type, it only performs a certain operation.

Ideally, we would have multiple workers, where each worker would be responsible for a certain area, and we would chain the workers to run periodically (for example, first we upload the holdings, if it succeeds, we upload the owners and keepers, and so on).

Initially, we had four download workers, DownloadConfigsWorker, DownloadHoldingsWorker, DownloadPeopleWorker and DownloadAnimalsWorker (and similar for upload workers). The goal was to chain them and run them periodically. However, it is not possible to chain periodic request, only one-time requests. Since it is crucial both for the downloading and uploading data that the requests stop running if the previous worker fails, to prevent inconsistent data, we had to use other approach. For example, if the order of the workers is DownloadConfigsWorker -> DownloadHoldingsWorker -> DownloadPeopleWorker -> DownloadAnimalsWorker, and the DownloadHoldingsWorker fails, if DownloadPeopleWorker runs and succeeds we would have people associated with a holding that does not exist (in the local database), which is not possible (i.e., invalid data).

To fix this problem, the download workers and upload workers were merged into two workers: DownloadWorker and UploadWorker. Instead of a (Work Manager) request having chained workers, each worker chains the API requests internally, which simulates the initial approach. However, this way we can run the Work Manager requests periodically and still stop running if any of the API requests fails, ensuring data consistency.

Together Room, DAOs, Coroutines, Retrofit, and Work Manager work together to form the Data Layer. With Room, we can have a local database, allowing the user to work while offline, also providing many useful functionalities, such as live updates of the data displayed. With DAOs, we can access Room in an intuitive way. With Coroutines, we can process CPU-heavy requests and I/O requests (such as certain Room accesses) asynchronously, in a synchronous way making the app easier to develop and maintain. With Work Manager we can synchronize (both download and upload) the user data to and from the remote API, making sure the user is working with the most updated data, and that his data is being uploaded so other users can use it. Dagger2 is used to fulfil the repositories and service's dependencies. LiveData is the main communication form between the Data Layer and the Business Layer, and when used with Room, we can make sure the user is presented with the most updated data.

4.4. Requirements

The application's requirements were defined prior to its start. Some were a mix from the D-LITS's WebApp requirements (such as holdings, owners/keepers, animals CRUD) and previous company's mobile applications (such as intervening animals, external devices interaction, and automatic data synchronization).

4.4.1. Functional requirements

A functional requirement is a description of a functionality a software must have. It can be something big and complicated or something small and simple. They describe what a system must do, how to behave and react to use.

Functional requirements should be gathered before the functionality is developed, since they will help the developers to make sure they are on track with the intended purpose of the application. They also help with catching errors before the actual development, which means said errors are easier and cheaper to fix. A detailed description of each user story and corresponding acceptance criteria is available in Appendix C – Functional Requirements – User Stories and Acceptance Criteria.

Regarding the major features related to the requirements, the architecture should contemplate the following:

- Login/Logout
- Enter/Update entity/brigade
- Holding CRUD
- Owner/Keeper CRUD
- Animal CRUD
- Service
- Summaries

- Devices Library
- Automatic synchronization
- Automatic seed of the local database

A more detailed explication of the functional requirements can be consulted in Appendix C – Functional Requirements - User Stories and Acceptance Criteria

4.4.2. Non-functional requirements

Non-functional requirements are used to measure the usability and effectiveness of the system, as well as the overall quality of the system: how fast it is, how much users and data it can handle, how resilient it is to bad inputs, how long it takes to develop and how much it costs, amongst other things. A system may be very good in a functionality aspect (lots of features that work as they should) but the system might be unusable (slow, hard to comprehend, no feedback), which ultimately makes the system useless.

Non-functional requirements can help the verification and validation process of, amongst other things, performance, stress, usability, security, feedback to invalid inputs. Said tests are made after the functional tests, to make sure the functionalities developed are usable.

Non-functional requirements are many times overlooked. Unexperienced developers usually focus a lot on if it works, and not on how well it works. When it comes to non-enterprise applications, for example, Instagram™, poor non-functional requirements (such as response times or security breaches) may make the users simply not use the applications; in these cases, the users are the ones choosing whether to use the application or not [42]. However, in enterprise applications, where the users are not the ones choosing whether to use the application or not, an application with poor non-functional requirements may make the users frustrated and uncooperative with the system.

When the same users are using the same application all day during a workday, it is very easy to get frustrated. Trying to work with a slow system, inserting data considered invalid by the system and not getting feedback of why it is invalid, not understanding how to use it to produce the expected results, are common conditions that make a system hard to use. When this happens, users might stop performing certain actions.

When looking and the application's non-functional requirements, some stand out more than others, such as performance, scalability, usability, and modifiability.

Performance is the time the software takes to respond to events and process information. Scalability is the highest workloads under which the system can work while still meeting the performance requirements. When paired together, they measure how well the application will perform when the user base starts to grow. The data the application will handle will increase a great deal, since a lot of it comes from the WebApp, that has its own user base. Weak performance or scalability can easily make the application unusable.

Usability measures how intuitive and easy to use the application is, how fast the users learn, how quickly they achieve their goals, amongst other things. Usability is essential in any application, since if the users feel frustrated by not being able to understand/do something, they will not use the application. Since a good part of the application is related to data inputs, good feedback is very important, not only, so the users do not feel frustrated when not understanding what they are doing wrong, but also allow them to fix their erroneous inputs, to ultimately achieve their objectives.

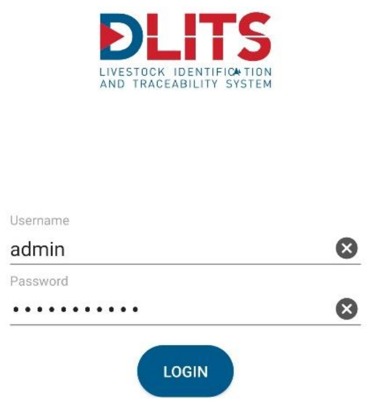
Modifiability can be described as how easy it is to modify the components to adapt to different scenarios. Clean code can be a big part of this, since if other developers cannot understand it, they will not be able to modify it. However, there is more to it than clean code. Since this is a generic architecture, the ability to adapt it to different scenarios is very important. It is crucial that when it is used to create a new application, the time it takes to detach and attach components should be as reduced as possible. Later we will talk about how we achieved this.

5. DLITS

This chapter talks about DLITS, the application which named this thesis. In this chapter, we will see the screens of the application, along with some of the technologies used to create them. We will also see some examples of fragments being reused, highlighting the reusability of the architecture.

5.1. Login

The login has two stages; the first one is the authentication, where the user must insert the credentials associated with his account. He must be linked to a veterinary or veterinary assistant to log in, shown in Figure 17.



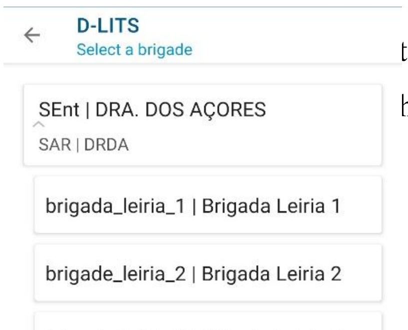
Username
admin
Password
•••••••••••

Figure 17 - Login

The Login button is an external library that allows the button to go into a loading state, notifying the user that something is being processed [43]. This also prevents the user from interacting with it while something is being processed.

The application uses Auth0 to decode the token coming from the remote API, which contains important, encrypted data [44].

Once he logs in, he must select the brigade he wants to start working on. He is presented with all the brigades he



D-LITS
Select a brigade

SEnt | DRA. DOS AÇORES
SAR | DRDA

brigada_leiria_1 | Brigada Leiria 1

brigade_leiria_2 | Brigada Leiria 2

is associated with, grouped by their respective entity, as shown in Figure 18.

Each brigade is associated with one or more entities. The application merges multiple Room requests to expose a list of “Entities” where each “Entity” has multiple brigades associated, to allow the interface to easily be constructed.

The entities and brigades are displayed using an ExpandableListView, a native Android view that shows items in a vertical scrolling two-level list [45].

Each item was custom created to display the most relevant information. In the entities, their unique identifier, and their name, with the respective role the user has in that entity below. In brigades, their unique identifier, and their name.

After selecting the brigade, he is directed to the Dashboard of the application.

5.2. Dashboard

The Dashboard is the main page of the application, the access point to most areas of the application, shown in Figure 19. The Dashboard has the primary actions available as big buttons, to facilitate its access, since it is what users usually will go to when they enter the application.

This Dashboard is a variation of Wezoot’s Dashboard, where the main actions are displayed on the main screen.

As the application grows, the Dashboard will be improved, with more relevant functionalities and icons, to better identify each functionality-

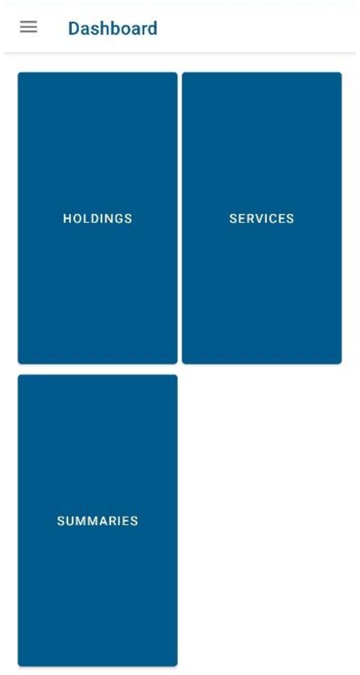
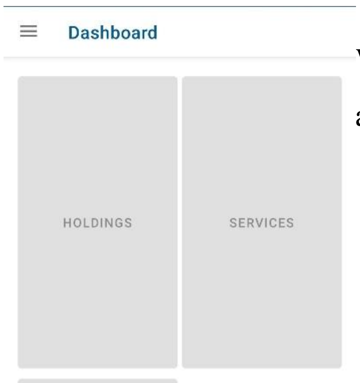


Figure 19 - Dashboard



When a user enters an entity and is online, the application automatically seeds the local database with the necessary data to operate, shown in Figure 20. While the data is being seeded, the user is prevented from progressing through the app, to prevent seeing invalid/inconsistent data. After the seed is done, the user is notified and free to use the rest of the app.

If the user is offline (must have that entity's data seeded previously) or the entity's data has been seeded not long ago, no seed is done. After the seed is complete (or if it is not necessary), the user can go offline.

The notification of the database being seeded was created using a Snackbar with a custom layout, with a ProgressBar to notify the progress of the seed [46], [47].

In the Dashboard, the user can also access the Main side menu, via the Burger menu on the top left corner. This menu holds, besides the main actions, secondary actions, such as Bluetooth and Wi-Fi settings, Change entity/brigade, Logout, and session information relevant to the user, such has the entity/brigade in which he is in, the user's role and his username, shown in Figure 21.

The side menu was created using a Drawer associated with the Navigation Library, where each item is associated with a destination. Some of the icons are directly from the Material Icons collections, others were custom made following the Material Design 2 guidelines [48] [49].

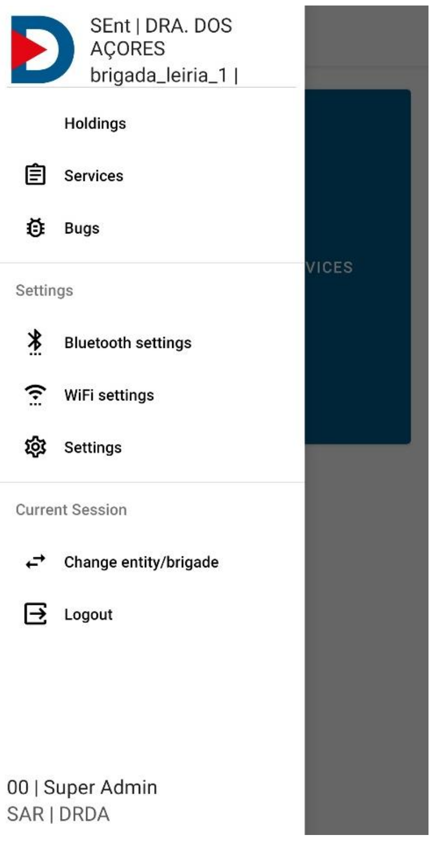
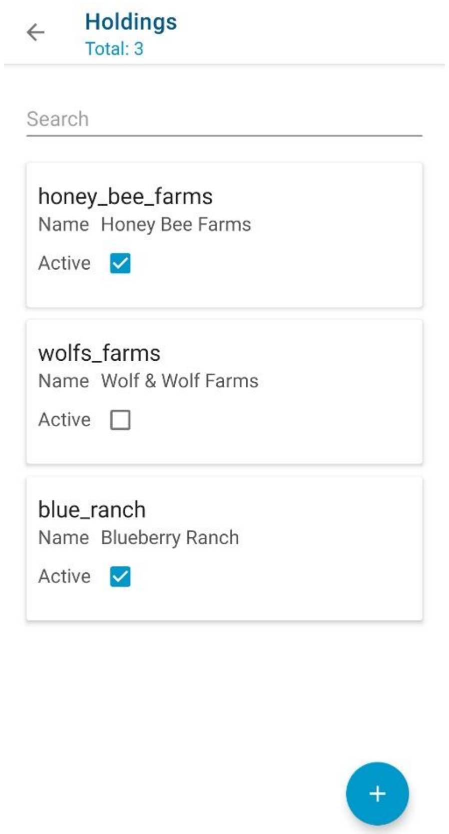


Figure 21 – Main side menu

5.3. Holdings

When the user accesses Holdings, he is presented with a list of all the holdings associated with the entity in which he is working in, shown in Figure 22. Each holding displays its unique identifier, its name, and its active state since this is the most relevant data for the user, displayed in a customized layout. If no holdings are available, a warning is displayed, using a Snackbar.

In the bottom right corner (called FloatingActionButton), a "create new" button (with an "+" icon) is displayed at the bottom of the screen, allowing the user to create a new holding [50]. If the has no permissions to create a new holding, the button is invisible. This "create new" button follows the Material Design 2 guidelines, which state the FloatingActionButton should be used for constructive actions, such as create.


Holdings
Total: 3
Search

honey_bee_farms
Name Honey Bee Farms
Active ✓

wolfs_farms
Name Wolf & Wolf Farms
Active ☐

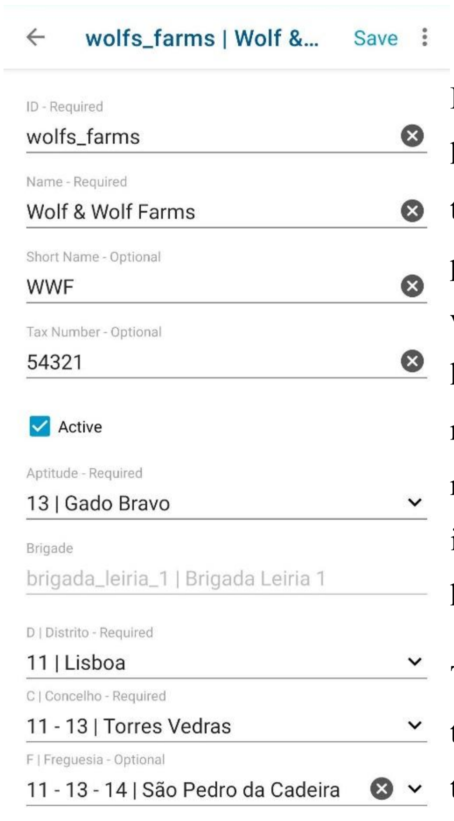
blue_ranch
Name Blueberry Ranch
Active ✓

+

Figure 22 - Holdings list

On the top of the page, it is also displayed where the user is, in this case, “Holdings”, along with the total holdings being listed.

A custom search functionality is also present; to facilitate the user finding the holding he is looking for. This custom edit text above the list is linked directly to the ViewModel that uses the input from the user to filter the holdings, displaying the ones that match the search, or all if no search is introduced. This is the default behavior for all search functionalities in the application.



ID - Required	wolfs_farms
Name - Required	Wolf & Wolf Farms
Short Name - Optional	WWF
Tax Number - Optional	54321
✓ Active	
Aptitude - Required	13 | Gado Bravo
Brigade	brigada_leiria_1 | Brigada Leiria 1
D | Distrito - Required	11 | Lisboa
C | Concelho - Required	11 - 13 | Torres Vedras
F | Freguesia - Optional	11 - 13 - 14 | São Pedro da Cadeira

Figure 23 - Holding's details

By clicking in a holding, the user is presented with the holding’s details, where he can update them, if he has the permissions, as shown in Figure 23. If he has no permissions to update, he can only consult the details, without changing them. In this page, its present the holding’s unique identifier, name, short name, tax number, active state, default aptitude of its animals, responsible brigade, and its country zone (configurable in the WebApp). The user can also create a new holding, by filling the same fields listed above.

The user can access its animals, owners, and keepers via the top right menu. Like in all places where it is possible to save, the save button is present on the top right corner of the screen.

Every editor displayed in this screen, such as the edit texts, the checkbox, and the dropdown menus, were custom created for this application, to support extra behaviors, such as clearing the content, displaying errors when related to that editor, regex (certain fields, such as the unique identifier, must follow rules), provide extra feedback to the user, amongst others. These editors, and others, are present in the entire application.

5.4. Owners/Keepers

When checking a holding's details, the user can check its current owners and keepers, by accessing the top right menu. This will direct the user to the holding's owners/keepers, where he can see the owners and the keepers that are currently working there, shown in Figure 24.

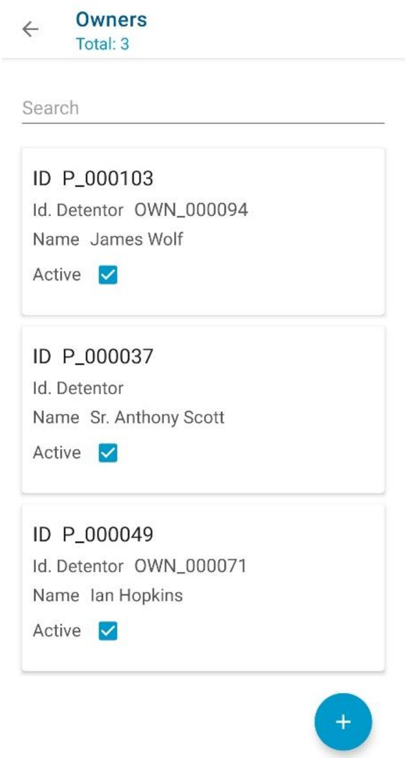

Owners
Total: 3
Search
ID P_000103
Id. Detentor OWN_000094
Name James Wolf
Active ✓
ID P_000037
Id. Detentor
Name Sr. Anthony Scott
Active ✓
ID P_000049
Id. Detentor OWN_000071
Name Ian Hopkins
Active ✓

Figure 24 - Owners list

The user is presented with a list of either owners or keepers (depending on what he chose before). Each owner/keeper displays its unique identifier as a person, its unique identifier as an owner/keeper (what identifies him professionally), and its active state, displayed in a custom layout. The user can create a new owner/keeper via the "create new" button (with an "+" icon), in the bottom right corner of the page.

Like in most listings, a search functionality is displayed above the list, connected directly to the ViewModel.

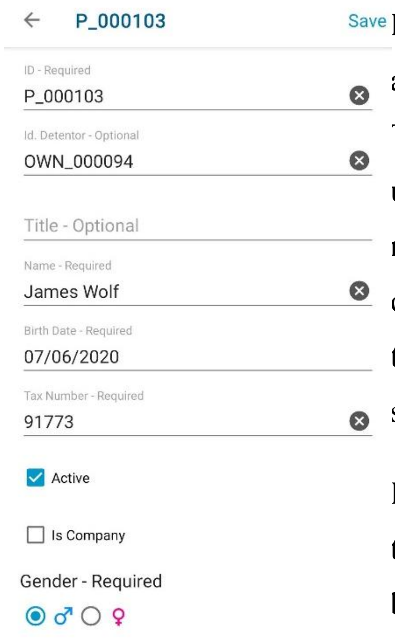

P_000103
Save
ID - Required
P_000103
Id. Detentor - Optional
OWN_000094
Title - Optional
Name - Required
James Wolf
Birth Date - Required
07/06/2020
Tax Number - Required
91773
✓ Active
☐ Is Company
Gender - Required
● ♂ ○ ♀

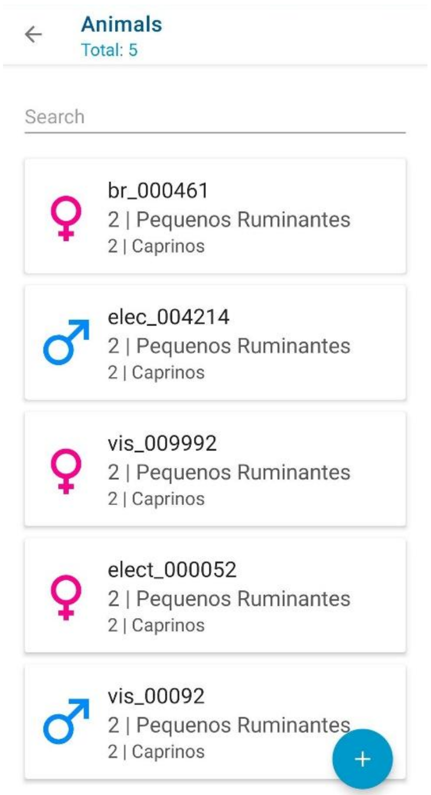
Figure 25 - Owner's details

By clicking an owner/keeper, the user can check its details and edit them (if he has permissions), shown in Figure 25. The user can edit the owner/keeper's unique identifier, unique identifier as an owner/keeper, name, birthdate, tax number, active state, if it is a company (only applies to owners) and, if it is not a company, its gender. Since both the listing and the editing of owners and keepers are very similar, only the owners are displayed here.

Like the edit texts present here, the birth date component and the gender component were custom created, to encapsulate behaviors and support extra functionalities. In the case of the date component, it allows the choosing of a date and time, according to the used precision (can be precision to the date, hour, minute or second, depending on the need). In this case, only the date is relevant.

5.5. Animals

Much like owners and keepers, the animals present in a holding can be accessed via the top right menu when checking the holding's details. Animals can have multiple identifications, but only one preferred, which is the one used to refer to the animal. This identification, along with its group, species and gender, is used to identify the animals in their listing, displayed in a custom layout, as shown in Figure 26. By clicking an animal, the user can check its details.



Animals
Total: 5

Search

♀ br_000461
2 | Pequenos Ruminantes
2 | Caprinos

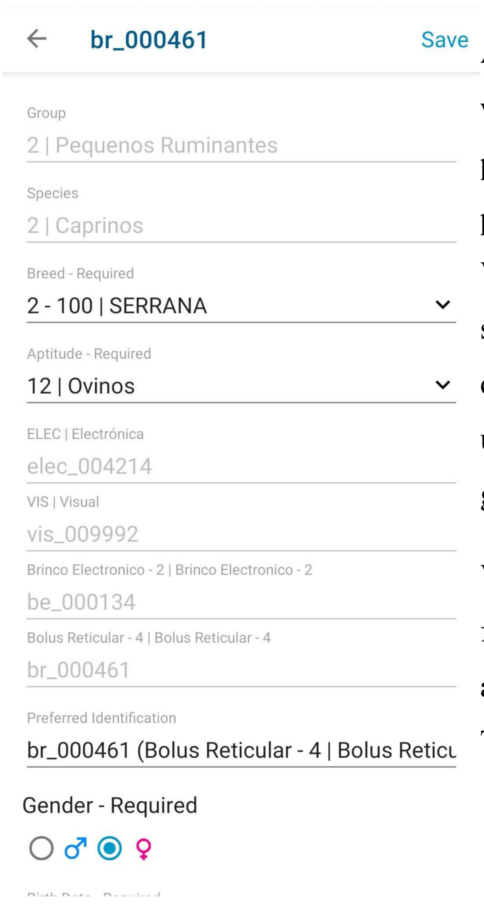
♂ elec_004214
2 | Pequenos Ruminantes
2 | Caprinos

♀ vis_009992
2 | Pequenos Ruminantes
2 | Caprinos

♀ elect_000052
2 | Pequenos Ruminantes
2 | Caprinos

♂ vis_00092
2 | Pequenos Ruminantes
2 | Caprinos

Figure 26 - Animals list



br_000461 Save

Group
2 | Pequenos Ruminantes

Species
2 | Caprinos

Breed - Required
2 - 100 | SERRANA

Aptitude - Required
12 | Ovinos

ELEC | Electrónica
elec_004214

VIS | Visual
vis_009992

Brinco Electronico - 2 | Brinco Electronico - 2
be_000134

Bolus Reticular - 4 | Bolus Reticular - 4
br_000461

Preferred Identification
br_000461 (Bolus Reticular - 4 | Bolus Reticu

Gender - Required
○ ♂ [✓ ♀]

Figure 27 - Animal's details

Amongst other data, we can see the animal's group, species (must be of the previously selected group), and breed (must be of the previously selected species), shown in Figure 27. When the selected group changes, the available species change too; when the selected species changes, the available breeds change too, so that the user can only select species belonging to the selected group, and breeds belonging to the selected species.

We can also see the animal's identifications (the four fields after the aptitude, configured in the WebApp), and the preferred identification (above the gender). The available types of identifications (in this case, "ELEC", "VIS", "Brinco Eletronico" and "Bolus Reticular") are created dynamically since they are configurable in the WebApp. When checking the animal's details, the identifications are always locked, since changing the identifications is a different, more complex process, which is yet to be implemented in the mobile version. However, by clicking the preferred identification component, the user is presented with a custom component and can change the preferred identification (i.e. select the new preferred identification amongst the ones already created.

Other relevant fields (not visible in this image), are the animal's mother and father and its owner and keeper (must be of the same holding, presented previously). The mother and father of the animal must be older than the animal, and of the same species. When the species of the animal, or its birth date, changes, the ViewModel filters and displays only the animals that fulfil these requirements, making sure the user is prevented only with valid data.

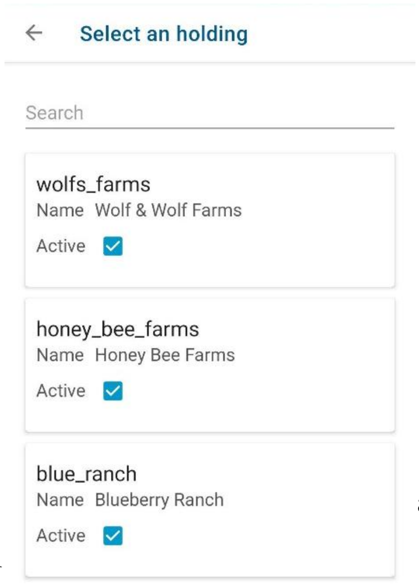
5.6.Services

Services are the focus of the application. Multiple animals, each having one or more actions being applied to it, compose a service. Brigades are in the fields identifying animals and performing actions on them, like applying vaccines, taking blood samples, etc.

5.6.1. Setup

Multiple actions can be performed to multiple animals at the same time; however, a service can only be constituted of animals of the same holding and the same group. So, before starting to perform actions, the user must select the holding in which he is operating, shown in Figure 28, and the group of animals to which he will be performing the actions on.

The component responsible for listing the available holdings is the same component previously shown, using a different data source, exemplifying the reusability of each component.


Select an holding

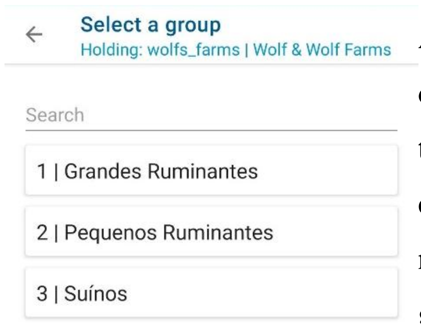
Search

wolfs_farms
Name Wolf & Wolf Farms
Active ✓

honey_bee_farms
Name Honey Bee Farms
Active ✓

blue_ranch
Name Blueberry Ranch
Active ✓

Figure 28 - Select the service's holding


Select a group
Holding: wolfs_farms | Wolf & Wolf Farms

Search

1 | Grandes Ruminantes

2 | Pequenos Ruminantes

3 | Suínos

Figure 29 – Select the service's group

After selecting the holding, the user must select the group of the animals to which the brigade is applying the actions to, shown in Figure 29. Although the animals are classified by group, species, and breed, only the group is relevant here, due to health procedures reasons (the same group of animals usually has the same diseases, and therefore, the same treatments, depending on the country's policies). Also, due to physical limitations, such as fences and the drafter gates adaptable size it is impractical since the size of animals from different groups and vary a lot).

After the group is selected, the user must select the date in which the actions are being/were performed (never after the current day), shown in Figure 30.

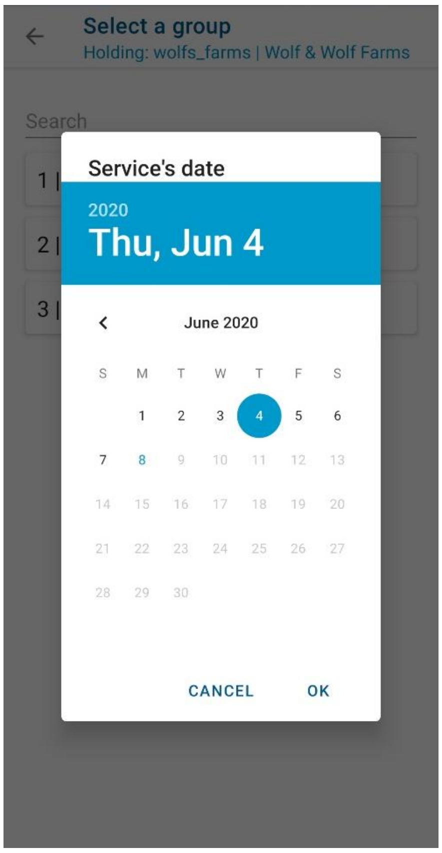


Figure 30 - Select the service's date

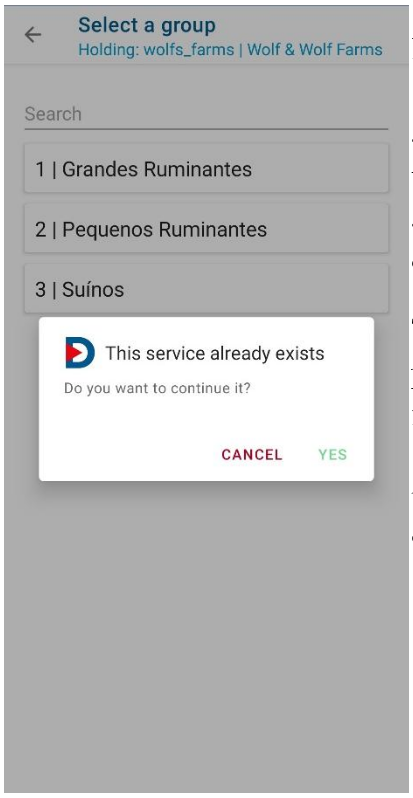

Select a group
Holding: wolfs_farms | Wolf & Wolf Farms
Search
1 | Grandes Ruminantes
2 | Pequenos Ruminantes
3 | Suínos
This service already exists
Do you want to continue it?
CANCEL
YES

Figure 31 - Ask user if he wants to continue the service

he
is

If a previously done service matches these parameters (same group, holding and date), and is yet to be finished, the user is asked if wants to continue it, shown in Figure 31. If the service already finished, the user can only consult without changing it.

The warning is displayed in an AlertDialog, since it is prioritary, and the user must pay attention to it, following the Material Design 2 guidelines [51].

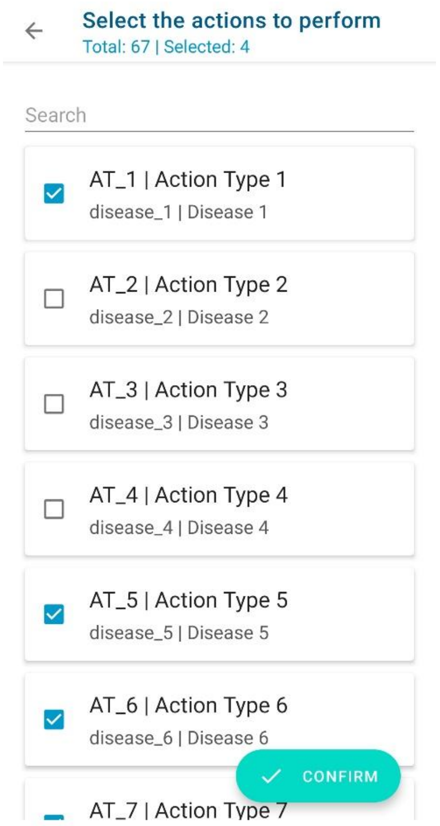
After this step, the user cannot change the parameters. Going back and choosing different data will start a new service (or continue other).

5.6.2. Service

After the setup is complete, the brigade can now start registering actions.

5.6.3. Action types list

Action types are the types of the actions that can be applied to the animals (usually called an intervention); an action is the application of a certain action type (for example, “Deworming” is the action type, and when it is applied to a certain animal, creates an action). Action types are configurable in the WebApp. To facilitate the application of actions in batch to multiple animals, the user can select a template of the actions types, which will be applied automatically to every animal he registers after (the user can change the template and the actions specific to any animal at any time, not influencing previously intervened animals). The list of available action types is shown in Figure 32.

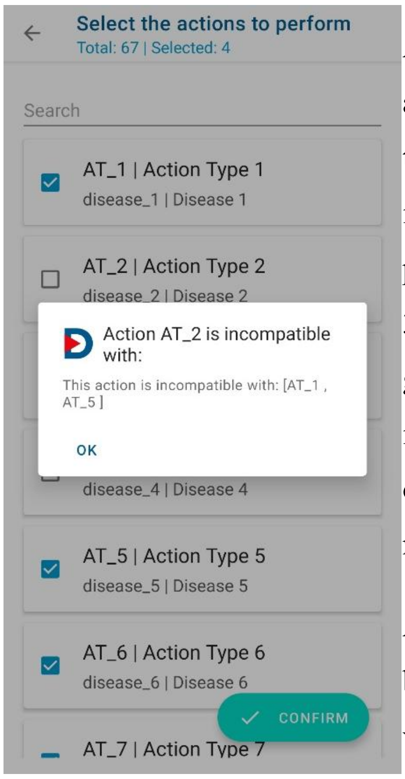

Select the actions to perform
Total: 67 | Selected: 4

Search

- [x] AT_1 | Action Type 1 — disease_1 | Disease 1
- [ ] AT_2 | Action Type 2 — disease_2 | Disease 2
- [ ] AT_3 | Action Type 3 — disease_3 | Disease 3
- [ ] AT_4 | Action Type 4 — disease_4 | Disease 4
- [x] AT_5 | Action Type 5 — disease_5 | Disease 5
- [x] AT_6 | Action Type 6 — disease_6 | Disease 6
- AT_7 | Action Type 7

Figure 32 - Action types list

Action types have multiple configurations, one of them is the animal group they can be applied to, which means only action types of the animal’s group can be applied to the animal. Each action type can have a disease associated, which would require a sample from the animal to be tested in laboratory.


Select the actions to perform
Total: 67 | Selected: 4

Search

AT_1 | Action Type 1
disease_1 | Disease 1

AT_2 | Action Type 2
disease_2 | Disease 2

Action AT_2 is incompatible with:

This action is incompatible with: [AT_1 , AT_5]

OK

disease_4 | Disease 4

AT_5 | Action Type 5
disease_5 | Disease 5

AT_6 | Action Type 6
disease_6 | Disease 6

CONFIRM

AT_7 | Action Type 7

Figure 33 - Incompatible action types

Action types can also be incompatible with each other. Two action types are incompatible if they cannot be applied at the same time to the same animal. When the user selects incompatible action types, a warning is displayed, preventing the user from selecting them, as shown in Figure 33. Following the Android separation of concerns guidelines, the ViewModel is notified when an action type is selected and makes sure that it is compatible with the ones selected. If it is incompatible, it notifies the view to notify the user of the incompatibility.

After the user is satisfied with his template, he can confirm by clicking the bottom right button. This button is a variation of the FloatingActionButton present throughout the application (usually with the "+" icon). It expands and shrinks as the user scrolls through the list, something that is seen in a lot of Google applications, such as Google Messages. When it is in the collapsed mode, it displays only the icon.

5.6.4. Intervened animals

The main screen of the service is the list of intervened animals. This list holds the animals that have at least one action being applied to them in this service.

Each animal displays its preferred identification, along with the unique identifier of each action being applied to them, separated by "|", as shown in Figure 34.

To intervene new animals, the user can click the bottom right bottom (that display a "+" when closed), which will present him with two options: one to connect to a RFID reader via Bluetooth, Wi-Fi or BLE, to a add them via the reader (the top option), and the other to add them manually (the middle option).

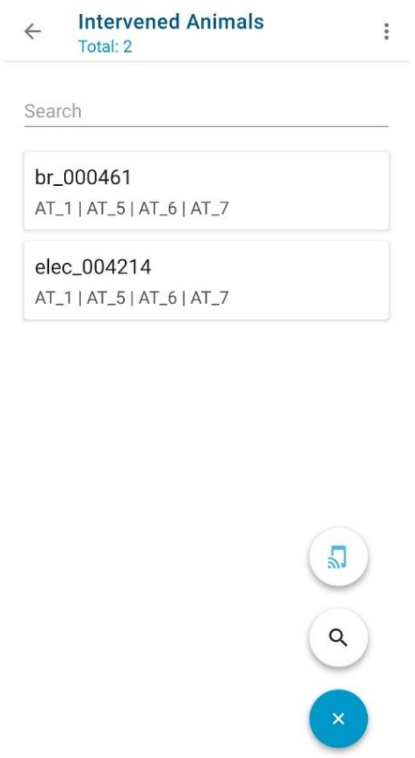
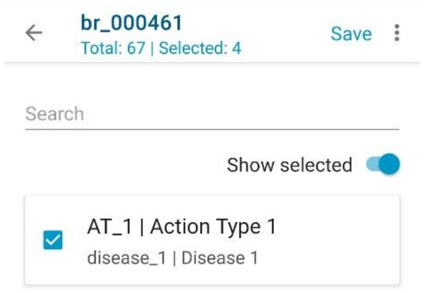

Intervened Animals
Total: 2
Search
br_000461
AT_1 | AT_5 | AT_6 | AT_7
elec_004214
AT_1 | AT_5 | AT_6 | AT_7

Figure 34 - Options to intervene animals

To remove an animal from the service (making it no longer intervened), the user can swipe it and confirm, using a custom layout. The animal will no longer be displayed in this list, but will be present in the non-intervened animals, shown further above. The swipe action is the default action to delete an item from a list in Android, seen in many applications, such as Outlook. The confirmation is necessary to prevent users from accidentally removing an animal from the service and having to do it all again.


br_000461
Total: 67 | Selected: 4
Save
Search
Show selected
AT_1 | Action Type 1
disease_1 | Disease 1

When the user selects one animal, he is presented with the actions being applied to that animal. By default, the option “show selected” is selected, which means the user only sees the actions being applied to the animal, as shown in Figure 35. This is to prevent the users from having to scroll a lot every time they want to check all the actions being applied to the animal.

This component is the same used to select the template action types. However, it is configured to this scenario, showing the reusability of the components.

By deselecting the “show selected” option, the user is presented with all the actions, as shown in Figure 36. If the user wants to apply new actions to the animal, he can deselect the option and add the new ones or remove previously selected ones (by checking and un-checking the checkboxes). Changing the actions being applied to an animal will not affect the template.

The user can use the search feature to search for specific actions to facilitate finding them. On the top, the user can save the actions being applied to the animal or check the animal’s details (via the “…” menu). The animal’s details are like the ones presented previously.

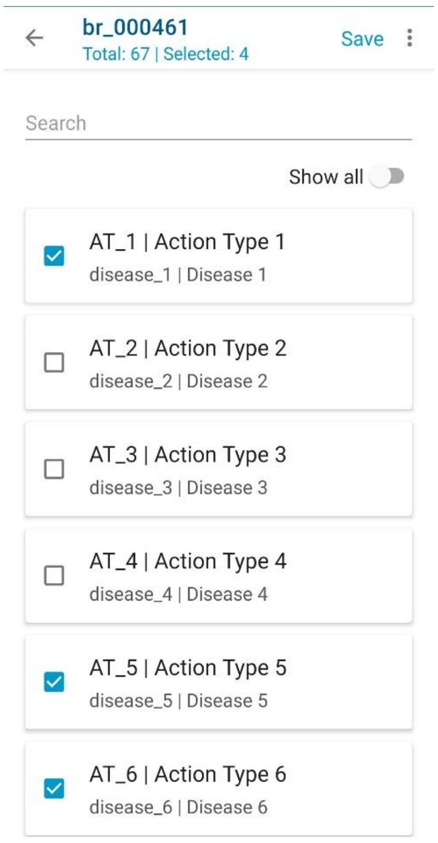
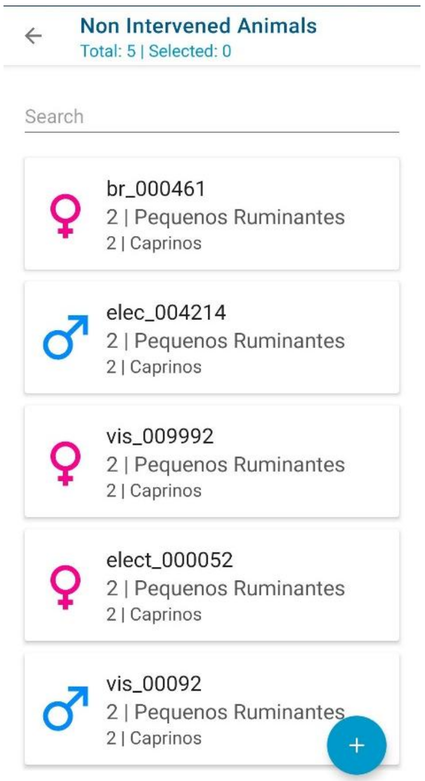

br_000461
Total: 67 | Selected: 4
Save
Search
Show all
✓ AT_1 | Action Type 1
disease_1 | Disease 1
☐ AT_2 | Action Type 2
disease_2 | Disease 2
☐ AT_3 | Action Type 3
disease_3 | Disease 3
☐ AT_4 | Action Type 4
disease_4 | Disease 4
✓ AT_5 | Action Type 5
disease_5 | Disease 5
✓ AT_6 | Action Type 6
disease_6 | Disease 6

Figure 36 - Actions available to apply to an aninal

5.6.5. Non-intervened animals

By selecting the middle option, the user is redirected to the non-intervened animals, that holds the animals of that group and holding, that are yet to be intervened in that service, as shown in Figure 37.

The user can also create a new animal using the bottom right button. The animal is automatically assigned the service's holding and group. This animal will not be automatically intervened, but will be associated with the service, which means that if the service is deleted (for whatever reason), so is the animal, since it was created in its context.



Non Intervened Animals
Total: 5 | Selected: 0

Search

br_000461
2 | Pequenos Ruminantes
2 | Caprinos

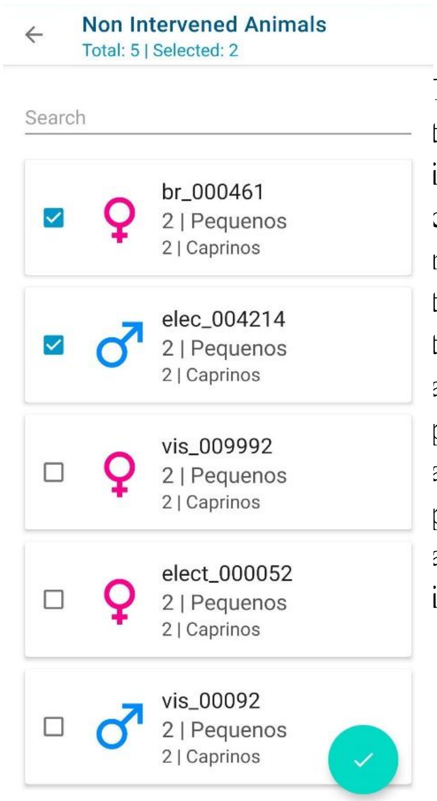
elec_004214
2 | Pequenos Ruminantes
2 | Caprinos

vis_009992
2 | Pequenos Ruminantes
2 | Caprinos

elect_000052
2 | Pequenos Ruminantes
2 | Caprinos

vis_00092
2 | Pequenos Ruminantes
2 | Caprinos

Figure 37 - Non-intervened animals



Non Intervened Animals
Total: 5 | Selected: 2

Search

br_000461
2 | Pequenos
2 | Caprinos

elec_004214
2 | Pequenos
2 | Caprinos

vis_009992
2 | Pequenos
2 | Caprinos

elect_000052
2 | Pequenos
2 | Caprinos

vis_00092
2 | Pequenos
2 | Caprinos

Figure 38 - Selecting animals to intervene

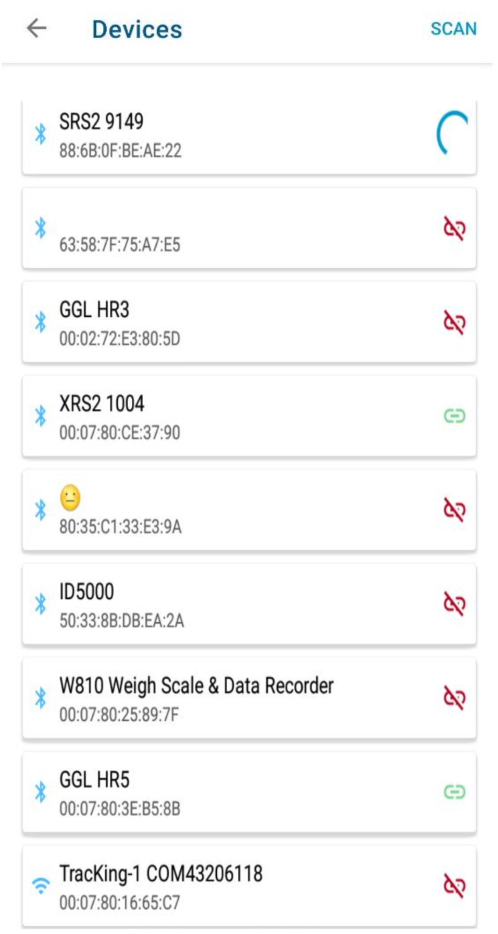
To check or edit an animal's details, the user can click the animal (while not in multi-selection mode), as shown in Figure 37. To select one, or more animals, the user can long click an animal; by doing this, it will enter multi-selection mode, as shown in Figure 38. By clicking the bottom right button, which now has a "check" sign, the user confirms the intervention of the selected animals, applying to them the template actions previously selected (he can still change individually the actions being applied to each animal, as shown previously). When the user confirms, the selected animals will no longer be on this list since they are now intervened.

5.6.6. Connecting to devices

By selecting the top option (the one with a device's icon) in Figure 34, the user is redirected to the devices page, where he is presented with multiple devices, either Bluetooth, BLE or Wi-Fi, that he can connect to, as shown in Figure 39.

When entering this page, it immediately starts scanning for devices. A scan can also be manually triggered, either via the top "Scan" button or by swiping down, which is the custom Android behavior to refresh the content of a list.

Each device displays, in a custom layout, its name (if any), address (depending on the device's type), and state: connecting, represented by a circular bar, as shown in the first device; not connect, as a red crossed symbol; connected, as a green symbol.


Devices
SCAN

SRS2 9149
88:6B:0F:BE:AE:22

63:58:7F:75:A7:E5

GGL HR3
00:02:72:E3:80:5D

XRS2 1004
00:07:80:CE:37:90

😑
80:35:C1:33:E3:9A

ID5000
50:33:8B:DB:EA:2A

W810 Weigh Scale & Data Recorder
00:07:80:25:89:7F

GGL HR5
00:07:80:3E:B5:8B

TracKing-1 COM43206118
00:07:80:16:65:C7

Figure 39 - Devices available to connect

Clicking a disconnected device's state symbol (shown as green) will attempt to connect it to the application. Clicking the red symbol will disconnect the device from the application.

This view is directly connected to the Devices Library: the devices available, the scan, connecting and disconnecting devices, are all functionalities provided by the library developed.

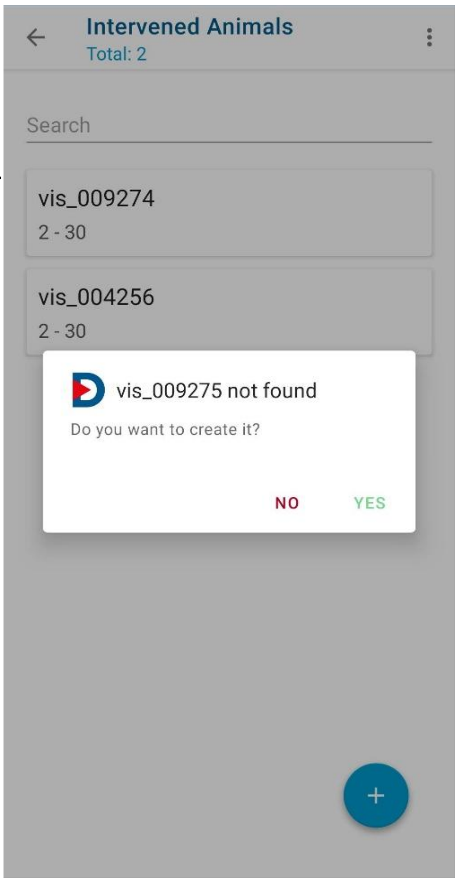
The application can be connected to more than one device, depending on the device type. After connecting to the desired devices, the user can go back to the service, and start using the devices. If the user has already previously connected with the desired devices, he does not need to connect to them again.

5.6.7. Reading identifications

Once the user returns to the service, he can start reading animals' identifications using the RFID readers (another functionality provided by the Devices Library). When the user reads an animal's identification, three scenarios might happen:

- The animal does not exist:

  If no animal of the service's group and holding has the identification read, the application asks the user if he wants to create the animal, as shown in Figure 40. If the user confirms, the applications starts the creation of the animal, the electronic identification automatically assigned as the identification read. The animal's group and holding are also automatically assigned as the service's. Once the user confirms the animal data, he must select the actions being applied to the animal (at least one). After the successful creation of the animal, it is now being intervened and associated with the service.



vis_009274
2 - 30
vis_004256
2 - 30
vis_009275 not found
Do you want to create it?

Figure 40 - Asking to create an animal from reading

- The animal is not being intervened:
  If an animal of the service's group and holding has the identification read, and the animal is yet to be intervened, it is automatically assigned the template actions, and will now be present in the list of intervened animals. The user is notified that the animal is now being intervened.
- The animal is already intervened:
  If an animal of the service's group and holding has the identification read, but that animal is already being intervened, the user is redirected to the animal's details, where he can change its actions and data (as shown previously).

After the user is satisfied with the service, he can finish it, via the top right menu at the service's main page. The service must have at least one animal to be finished. If the user leaves the service without intervening at least one animal, the service will not be created.

5.7. Summaries

In Summaries the user can see what he has performed in the current session, as shown in Figure 41. So far, it is only possible to check the services performed. Further ahead, the user will be able to see created/edit holdings, owners, keepers, and animals too.

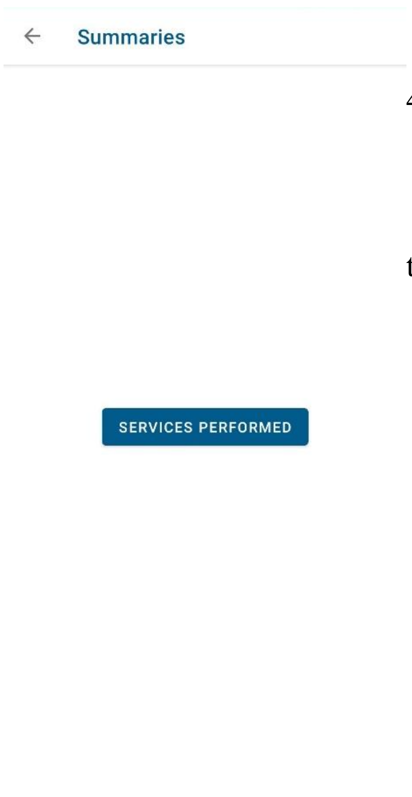
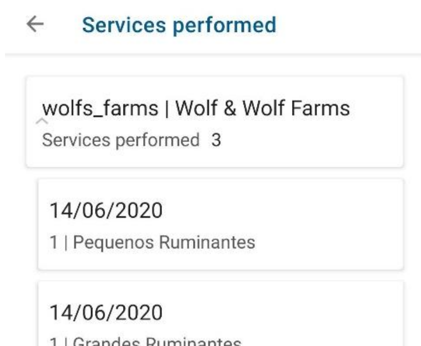


Figure 41 - Summaries



wolfs_farms | Wolf & Wolf Farms
Services performed 3

14/06/2020
1 | Pequenos Ruminantes

14/06/2020
1 | Grandes Ruminantes

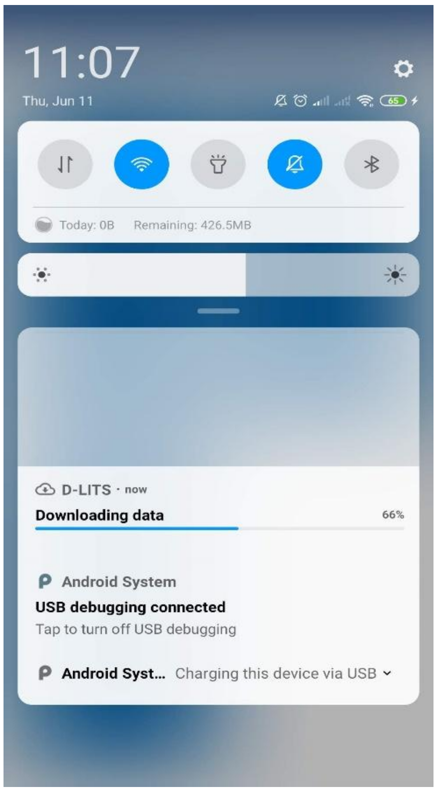
When listing the services performed, the user sees the services grouped by holding, as shown in Figure 42. Each service is represented by the date and group of the animals intervened. When clicking a service, the user is redirected to the service and can edit it (if it is yet to be synchronized) or consult it (if it is already synchronized).

5.8. Synchronization

Even though the user does not interact directly with the synchronization, since it is automatic, he is notified when the data is downloaded and uploaded, as shown in Figure 43. Once the data is successfully downloaded/uploaded, the user is also notified. Data will be downloaded/uploaded periodically when the application sees necessary.

Conflicts of the uploaded data will need to be solved in the WebApp, by another user, who must choose which version of the data to keep.

To reduce the amount of conflicts with the downloaded data, the download will occur frequently, making sure the user is (most of the time) working with the most updated data. The upload period is bigger, and only done when necessary, this is, the user has data changes to upload.


11:07
Thu, Jun 11

Today: 0B Remaining: 426.5MB

D-LITS · now
Downloading data 66%

Android System
USB debugging connected
Tap to turn off USB debugging

Android Syst... Charging this device via USB

Figure 43 - Notification of data being downloaded to the application

In the future, the application will also have its own conflict management system to solve conflicts of the user's data and the data download. For example, if the user creates a holding with the unique identifier "holding_0037" (that did not yet exist when he created it), and when the data is download, a holding with the same unique identifier comes, there will be a conflict, since two holdings cannot have the same unique identifier. The user will have to choose which version of the data to keep.

6. Critical analysis and improvement measures proposal

The planning and development phases went by smoothly. The fact the mobile application is to support a previously developed WebApp (which has all the requirements, functionalities, and data models consolidated) allowed the planning phase to be shorter than what it would have been otherwise. The company's experience in this field of work allowed for lots of usability issues to be caught and solved without the need of testers, potentially shortening the test phase.

The biggest problem encountered was creating the right architecture as soon as possible, since the later it would be achieved, the harder it would be to implement in the entire application. When one architecture seemed to fit the desired goals, some obstacle would appear which rendered the architecture unviable. With each version created, complexity was added.

The creation of a generic, flexible, and reactive architecture was achieved. However, it is not perfect. Each fragment (and their respective ViewModel) can be adapted to multiple scenarios, allowing their reusability without the need to create an extra, similar component for each specific usage. Even though this greatly speeds up the development process (the first usage might take a while, but the consecutive ones take significantly less time), it also adds complexity to each component. Instead of a component having a specific behavior, it must adapt to each case, depending on the controlling component. Having multiple, similar components would make each component easier to develop and change. However, this change could need to be reproduced through the multiple similar components, depending on the nature of the change. Each extra adaptation adds complexity and could possibly bring bugs to the other usages (a change to fit one scenario having unexpected behaviors in others). There needs to be a balance between creating generic, flexible components, that can adapt to different scenarios, and creating components so flexible that they take a long time to develop and adapt. This balance needs to be studied; before developing a component, it is necessary to try to predict (within reason) the amount of flexibility the component will need. Although, it was not the main cause for the delays, in the future it is necessary to not overengineer the generic components that could delay the development unnecessarily.

Other difficulty faced in terms of development was handling LiveData. Due to its nature, when being used, LiveData can become hard to handle, since sometimes events are unexpectedly triggered, causing the need to carefully handle each possible scenario (the current state of the component when the event is triggered). This is not really a problem of the architecture, but rather a difficulty of the framework. The same problem would have happened with a different architecture (as it has happened with the previous versions of the architecture) but could have been easier to handle, due to the complexity this architecture adds.

As mentioned previously, the usability of the application is very important, and to test this, usability tests are necessary, with people outside the development (ideally real users). However, due to the late ending of the development phase, the usability tests were delayed. In junction with the pandemic outbreak, which caused the company to work remotely, the usability tests were pushed further as the people that would normally perform the tests were overburden with extra work. This caused the usability tests to have been performed in lower quantity and by a smaller group of users, which could give a not so accurate representation of the target group. In the future, it is necessary to perform the usability tests sooner and with a bigger tester group to allow room for more accurate results and fixes. Also, each usability test round that is made should be link to a questionnaire, to better evaluate the usability.

Although the design of the interface was not a huge priority, we tried to create a clean interface, balancing the amount of information presented to the user while trying to not saturate the screen. We also tried to give as much feedback as possible to give the under to the right path. However, there is still room for improvements, some of them might come from the usability tests, such as prioritizing some information over other, or improving the feedback given. The input of a designer could greatly improve the interface, both in making it better looking and more usable.

Even though some parts took longer than expected (such has the initial development phase), the company's flexibility allowed to compensate the time "lost". The frequent meetings and demos allowed for improvements to be implemented sooner, which would be more difficult to implement later. The knowledge passed allowed for lots of mistakes to be avoided, since someone encountered them previously. The good communication allowed for lots of feedback which resulted in small, regular improvements.

7. Conclusion

It was proposed the development of a native Android application with a generic architecture that could be used to speed up the development of future similar applications. This goal was achieved, creating an application with a flexible architecture that could, not only, adapt to new scenarios, but also be reused to new similar applications. The library developed to abstract the communication with RFID devices can be used separately to this application, standardizing the communication with devices amongst the company's applications.

To achieve the result, a lot of research was made, to assure the application followed the Android guidelines while being versatile, easy to adapt, both to different scenarios and to future applications. Multiple versions of the architecture were developed and tested, until one that better fits the goals was achieved.

The internship allowed the deepening of the knowledge in Android, the recommended guidelines, and the technologies being released to help developers create better applications. When paired with this thesis, it allowed the connection between the practical side, performed during the internship, and the theoretical side, learned both during the first year of the masters and during the internship.

The initial weeks of the internship were somewhat troublesome, trying to figure out the right architecture, which took longer than expected. After that, the development went by smoothly, with the company taking a great part in it, primarily in knowledge sharing. The fact that there already was a remote API to support the WebApp, previously developed, also helped reduce the development time, as it would take a good amount of time to setup the remote API. Since it was already created, it was just necessary adapt it to the application's necessities. The main problems encountered were in finding the right architecture; when one architecture seemed to be good, something would show up (usually related to the GUI, communication between fragments and activities, amongst others), which the architecture failed to support. This would make the architecture design process restart. Each iteration would cause the changes to be applied to the entire application, delaying the development. Despite the time spent on developing multiple architecture iterations, the fact that I was already in the company before the internship knowing all the business logic associated with the project, avoided the usual time spent on the initial adaptation phase, which in turn eased the development process.

A generic architecture could greatly improve the development of future similar applications. Developers will spend less time creating the base of the project and more time on what makes it unique, adapting it to the client's wishes. If the base architecture keeps getting improved with each new application, the benefits will be even greater. The improvement of the Devices Library will also improve each application that uses it, making the effort much more beneficial.

As the next steps to take, it is important to focus on the usability tests; wait for the results on the ones being performed, evaluated, and improve the application accordingly. This is something that will be done multiple times throughout the development of the application since there is always room from improvement. As for features, a conflict management system within the application will be implemented, to allow the users to, when a conflict appears, keep their data, or use the one coming from the remote API. The Devices Library will also be further developed, to allow more and more devices which require specific implementations.

Bibliography

[1] “Digidelta-Software.” [Online]. Available: https://www.digidelta-software.com/. [Accessed: 26-Dec-2019].

[2] “Digidelta-Software | Projetos.” [Online]. Available: https://www.digidelta-software.com/projetos/identificacao-e-rastreabilidade-animal/pisa-net. [Accessed: 26-Dec-2019].

[3] “Home - Datamars.” [Online]. Available: https://datamars.com/. [Accessed: 24-May-2020].

[4] “Digidelta-Software | Equipamentos.” [Online]. Available: https://www.digidelta-software.com/equipamentos/brincos-/brincos-brincos-eletronicos. [Accessed: 25-May-2020].

[5] “Digidelta-Software | Equipamentos.” [Online]. Available: https://www.digidelta-software.com/equipamentos/leitores-e-balancas-rfid/leitores-e-balancas-rfid-leitor-ges3s. [Accessed: 25-May-2020].

[6] “Amfratech – making life simpler.” [Online]. Available: https://amfratech.com/. [Accessed: 17-Jun-2020].

[7] “Home | Food and Agriculture Organization of the United Nations.” [Online]. Available: http://www.fao.org/home/en/. [Accessed: 17-Jun-2020].

[8] “| ICAR.” [Online]. Available: https://www.icar.org/. [Accessed: 17-Jun-2020].

[9] “Windows Forms | Microsoft Docs.” [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/winforms/. [Accessed: 01-Apr-2020].

[10] “wezoot – zootechnics expertise – Wezoot – Gestão da Produção Animal. Maximize a sua produção!” [Online]. Available: https://wezoot.com/. [Accessed: 12-Apr-2020].

[11] “Yagro – Aplicações no Google Play.” [Online]. Available:

https://play.google.com/store/apps/details?id=com.yagro.yagroandroidapp&pcampaignid=MKT-Other-global-all-co-prtnr-py-PartBadge-Mar2515-1. [Accessed: 25-May-2020].

[12] "Herdwatch Farm & Herd App | Simplifying Farming – Aplicações no Google Play." [Online]. Available: https://play.google.com/store/apps/details?id=com.frs.hwprodie. [Accessed: 25-May-2020].

[13] "FarmWizard – Aplicações no Google Play." [Online]. Available: https://play.google.com/store/apps/details?id=com.farmwizard.FarmWizard. [Accessed: 25-May-2020].

[14] "Guide to app architecture | Android Developers." [Online]. Available: https://developer.android.com/jetpack/guide. [Accessed: 29-Jun-2020].

[15] "Activity | Android Developers." [Online]. Available: https://developer.android.com/reference/android/app/Activity. [Accessed: 14-Dec-2019].

[16] "Fragments | Android Developers." [Online]. Available: https://developer.android.com/guide/components/fragments. [Accessed: 14-Dec-2019].

[17] "LiveData Overview | Android Developers." [Online]. Available: https://developer.android.com/topic/libraries/architecture/livedata. [Accessed: 23-Nov-2019].

[18] "ViewModel Overview | Android Developers." [Online]. Available: https://developer.android.com/topic/libraries/architecture/viewmodel. [Accessed: 06-Feb-2020].

[19] "Data Binding Library | Android Developers." [Online]. Available: https://developer.android.com/topic/libraries/data-binding. [Accessed: 08-Feb-2020].

[20] "Two-way data binding | Android Developers." [Online]. Available: https://developer.android.com/topic/libraries/data-binding/two-way. [Accessed: 11-

Jun-2020].

[21] “Homepage - Material Design.” [Online]. Available: https://material.io/. [Accessed: 16-Dec-2019].

[22] “Develop for Android - Material Design.” [Online]. Available: https://material.io/develop/android/. [Accessed: 16-Dec-2019].

[23] “Navigation | Android Developers.” [Online]. Available: https://developer.android.com/guide/navigation#top_of_page. [Accessed: 29-Apr-2020].

[24] “Principles of navigation | Android Developers.” [Online]. Available: https://developer.android.com/guide/navigation/navigation-principles. [Accessed: 29-Apr-2020].

[25] “Dagger.” [Online]. Available: https://dagger.dev/. [Accessed: 16-Dec-2019].

[26] “Bluetooth® Technology Website.” [Online]. Available: https://www.bluetooth.com/. [Accessed: 12-Jun-2020].

[27] “Wi-Fi Alliance.” [Online]. Available: https://www.wi-fi.org/. [Accessed: 12-Jun-2020].

[28] “ReactiveX.” [Online]. Available: http://reactivex.io/. [Accessed: 23-Nov-2019].

[29] “GitHub - ReactiveX/RxJava: RxJava – Reactive Extensions for the JVM – a library for composing asynchronous and event-based programs using observable sequences for the Java VM.” [Online]. Available: https://github.com/ReactiveX/RxJava. [Accessed: 23-Nov-2019].

[30] “GitHub - ReactiveX/RxKotlin: RxJava bindings for Kotlin.” [Online]. Available: https://github.com/ReactiveX/RxKotlin. [Accessed: 23-Nov-2019].

[31] “TypeConverter | Android Developers.” [Online]. Available: https://developer.android.com/reference/android/arch/persistence/room/TypeConverter. [Accessed: 01-Dec-2019].

[32] “Run-Time Text Generation with T4 Text Templates - Visual Studio | Microsoft Docs.” [Online]. Available: https://docs.microsoft.com/en-

us/visualstudio/modeling/run-time-text-generation-with-t4-text-templates?view=vs-2019. [Accessed: 14-Dec-2019].

[33] “Dao | Android Developers.” [Online]. Available: https://developer.android.com/reference/androidx/room/Dao. [Accessed: 16-Mar-2020].

[34] “Coroutines Overview - Kotlin Programming Language.” [Online]. Available: https://kotlinlang.org/docs/reference/coroutines-overview.html. [Accessed: 19-Feb-2020].

[35] “Improve app performance with Kotlin coroutines | Android Developers.” [Online]. Available: https://developer.android.com/kotlin/coroutines#main-safety. [Accessed: 19-Feb-2020].

[36] “Retrofit.” [Online]. Available: https://square.github.io/retrofit/. [Accessed: 06-Feb-2020].

[37] “Square Open Source.” [Online]. Available: https://square.github.io/. [Accessed: 16-Dec-2019].

[38] “OkHttp.” [Online]. Available: https://square.github.io/okhttp/. [Accessed: 10-Jun-2020].

[39] “HTTP | MDN.” [Online]. Available: https://developer.mozilla.org/pt-PT/docs/Web/HTTP. [Accessed: 10-Jun-2020].

[40] “GitHub - google/gson: A Java serialization/deserialization library to convert Java Objects into JSON and back.” [Online]. Available: https://github.com/google/gson. [Accessed: 15-Mar-2020].

[41] “Schedule tasks with WorkManager | Android Developers.” [Online]. Available: https://developer.android.com/topic/libraries/architecture/workmanager. [Accessed: 11-May-2020].

[42] “Instagram.” [Online]. Available: https://www.instagram.com/. [Accessed: 12-Jun-2020].

[43] “GitHub - leandroBorgesFerreira/LoadingButtonAndroid: A button to substitute the

ProgressDialog.” [Online]. Available: https://github.com/leandroBorgesFerreira/LoadingButtonAndroid. [Accessed: 30-Jun-2020].

[44] “Auth0: Secure access for everyone. But not just anyone.” [Online]. Available: https://auth0.com/. [Accessed: 30-Jun-2020].

[45] “ExpandableListView | Android Developers.” [Online]. Available: https://developer.android.com/reference/android/widget/ExpandableListView. [Accessed: 30-Jun-2020].

[46] “Snackbar | Android Developers.” [Online]. Available: https://developer.android.com/reference/com/google/android/material/snackbar/Snackbar. [Accessed: 30-Jun-2020].

[47] “ProgressBar | Android Developers.” [Online]. Available: https://developer.android.com/reference/android/widget/ProgressBar. [Accessed: 30-Jun-2020].

[48] “Icons - Material Design.” [Online]. Available: https://material.io/resources/icons/?style=baseline. [Accessed: 02-Jul-2020].

[49] “Android icons - Material Design.” [Online]. Available: https://material.io/design/platform-guidance/android-icons.html#usage. [Accessed: 02-Jul-2020].

[50] “Buttons: floating action button - Material Design.” [Online]. Available: https://material.io/components/buttons-floating-action-button. [Accessed: 30-Jun-2020].

[51] “Dialogs - Material Design.” [Online]. Available: https://material.io/components/dialogs#usage. [Accessed: 30-Jun-2020].

[52] “JavaScript | MDN.” [Online]. Available: https://developer.mozilla.org/pt-PT/docs/Web/JavaScript. [Accessed: 28-Mar-2020].

[53] “HTML: Linguagem de Marcação de Hipertexto | MDN.” [Online]. Available: https://developer.mozilla.org/pt-PT/docs/Web/HTML. [Accessed: 28-Mar-2020].

[54] "CSS: Folhas de Estilo em Cascata | MDN." [Online]. Available: https://developer.mozilla.org/pt-PT/docs/Web/CSS. [Accessed: 28-Mar-2020].

[55] "React Native · A framework for building native apps using React." [Online]. Available: https://reactnative.dev/. [Accessed: 28-Mar-2020].

[56] "React – A JavaScript library for building user interfaces." [Online]. Available: https://reactjs.org/. [Accessed: 28-Mar-2020].

[57] "Ionic - Cross-Platform Mobile App Development." [Online]. Available: https://ionicframework.com/. [Accessed: 28-Mar-2020].

[58] "Native mobile apps with Angular, Vue.js, TypeScript, JavaScript - NativeScript." [Online]. Available: https://www.nativescript.org/. [Accessed: 28-Mar-2020].

[59] "Android Developers." [Online]. Available: https://developer.android.com/design. [Accessed: 29-Mar-2020].

[60] "Themes - iOS - Human Interface Guidelines - Apple Developer." [Online]. Available: https://developer.apple.com/design/human-interface-guidelines/ios/overview/themes/. [Accessed: 29-Mar-2020].

[61] S. Xanthopoulos and S. Xinogalos, "A comparative analysis of cross-platform development approaches for mobile applications," in *ACM International Conference Proceeding Series*, 2013, pp. 213–220.

[62] "Mobile Operating System Market Share Worldwide | StatCounter Global Stats." [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide. [Accessed: 29-Mar-2020].

[63] "Mobile Operating System Market Share Portugal | StatCounter Global Stats." [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/portugal. [Accessed: 29-Mar-2020].

[64] "Mobile Operating System Market Share United States Of America | StatCounter Global Stats." [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/united-states-of-america. [Accessed: 29-Mar-2020].

[65] "JetBrains: Developer Tools for Professionals and Teams." [Online]. Available:

https://www.jetbrains.com/. [Accessed: 08-Mar-2020].

[66] “O que é o Java?” [Online]. Available: https://www.java.com/pt_BR/about/whatis_java.jsp. [Accessed: 10-Jun-2020].

[67] “Comparison to Java - Kotlin Programming Language.” [Online]. Available: https://kotlinlang.org/docs/reference/comparison-to-java.html. [Accessed: 25-Feb-2020].

[68] “Google Play.” [Online]. Available: https://play.google.com/store?hl=pt-PT. [Accessed: 10-Jun-2020].

[69] “Exceptions: try, catch, finally, throw, Nothing - Kotlin Programming Language.” [Online]. Available: https://kotlinlang.org/docs/reference/exceptions.html. [Accessed: 25-Feb-2020].

[70] “Effective Java.” [Online]. Available: https://www.oracle.com/technetwork/java/effectivejava-136174.html. [Accessed: 25-Feb-2020].

[71] “Basic Types: Numbers, Strings, Arrays - Kotlin Programming Language.” [Online]. Available: https://kotlinlang.org/docs/reference/basic-types.html. [Accessed: 25-Feb-2020].

Appendices

Appendix A - Technology Selection

There are multiple languages, platforms, and frameworks to get to different solutions. Each one has their pros and cons; it is the developer's job to decide which one better fits its needs.

There are a lot of frameworks for mobile app development. If the developer is new to mobile development and has a good web development background, there are a lot of frameworks that use web technologies, such as JavaScript, HTML and CSS, to develop mobile apps [52], [53], [54].

- React Native [55]
  React Native is a framework created by Facebook in 2015. It uses React, which is a JavaScript library used to builder user interfaces [56]. React Native is cross-platform, which means it is possible to compile the same application for both Android, iOS and web. This means that the developers only need to develop one application for multiple devices.
- Ionic [57]
  The Ionic Framework is an open-source UI toolkit for building mobile and desktop applications using web technologies. Like React Native, Ionic is a cross-platform framework.
- NativeScript [58]
  NativeScript is a cross-platform framework for building iOS and Android applications using JavaScript and CSS. Its views are rendered using the native platform's rendering engine instead of using WebViews, resulting in native-like performance and UX. The fact that NativeScript is cross-platform and its rendering engine means only one code base is used for both Android and iOS, while still having the native behaviour and performance, the best of both worlds.

Cross-platform apps have the great upside of only one code base being necessary for both operating systems. When the company wants the app for both Android and iOS, this can greatly reduce the time-to-market, the amount of personnel needed to develop and maintain it, and the set of skills needed (JavaScript, HTML and CSS for both versions vs the technologies associated with each native version).

Even though that when using a cross-platform framework only one application needs to be developed to run on both OSs, it has the downside of having twice as many problems to deal with. Considering that both Android and iOS have their guidelines, the same application would have to follow both their guidelines, adapting its interface for either Android or iOS, depending on the device [59] [60]. Also, not always the same exact code can be used for both OSs, sometimes it needs adaptation to fit either one [61]. These differences can lead to unexpected problems that happen on one OS and not on the other and finding these problems becomes harder. In addition, because extra code is needed to fit both OSs, the code can become complex, especially in bigger applications. Although cross-platforms frameworks are becoming better at preventing extra code necessities, they are not perfect yet. However, if it is required to have the same application on both OSs, unless the company has many resources to spend on developing and maintain both versions of the app (for the two OSs), cross-platform might be a good idea.

Cross-platform applications have their upsides, but so do native ones. Native applications usually have a better performance, both in speed and in resource usage. This comes from their specificity; while cross-platform ones are developed in a one-fits-all basis, native apps are specific for their OS. Native apps also have greater access to the device's APIs (Application Programming Interface), whether its storage, camera, communication technologies or GPS, and the ability to use them offline, which might not always be possible in non-native apps. Depending on the necessary resources that the app needs, this might be a deciding factor.

Other very important factor when deciding which platform to develop the app in, is the company's previous experience. Although native applications usually provide a better usability and performance, a good cross-platform app will most likely out-perform a poor native application, both in performance and in usability. If the company has no previous experience with native development, but has good background on web development, a cross-platform application might be a better choice. However, in this specific case, although the company has experience in web development, it also has experience in mobile development, especially in Android, which makes this a not so important factor.

Time-to-market is also very important. Native applications tend to take longer to develop compared to cross-platform ones, especially if it is necessary to develop two native ones (one for each OS) versus a single cross-platform one that runs on both operating systems. However, when the target is only one OS it might be better to develop a native application, whether for Android or iOS.

When choosing which one is the best OS it is important to know the purpose of the application, the target market, and most of the user's OS in the target country (assuming it's an application for a certain country). When seen from a global perspective, according to [62], 73.3% of the world's devices run Android, while only 25.9% run iOS. If we look at the same stats in Portugal, we can see they are relatively similar, with Android having a 79.7% presence while iOS having only 20.1% [63]. However, when we look at United States of America, we can see that these stats show a very different reality, with Android having only 41.3% presence while iOS has the majority with 58.4% [64]. Even though that Android has the majority worldwide, when developing an app for a specific country, it is relevant to check both OS's presence in that country, to help us decide which is the best OS to develop the app in.

When looking specifically at this application's case, the choice was clear. Although the app is not only for domestic use, Portugal's target market can be used as an example of what would be like in other countries. This application's target are the veterinarians and their assistants that form a brigade. Brigades work in the field performing actions to animals. Since the field is usually dirty, wet, and generally not the best environment for a delicate device, they usually prefer rugged smartphones. These devices are normal smartphones, except they are made to withstand unfavorable conditions, like the ones mentioned above. These smartphones usually have stronger screens and bodies, and low level of permeability to dust and water. They are also more expensive than a similar, not rugged device. Although there are hard covers for iOS devices, the offer for rugged iOS devices is lower and much more expensive, making Android devices the favorites. Considering the time-window to develop the app as well as the previous experience and the need only for an Android version, amongst other factors, it was decided that native Android would be the best option for the application.

For native Android, there are two possible languages: Java and Kotlin. Although Java has been around for longer, Kotlin has been gaining significant visibility in the market in the past few years. A study was made to decide between the two languages. Kotlin was the choice; in Appendix B – Kotlin vs Java, the study can be read in further detail.

Appendix B - Kotlin vs Java

Kotlin is a programming language for Java Virtual Machine (JVM) and Android created in 2011 by JetBrains [65][66]. Although Kotlin's syntax is not similar to Java's, Kotlin is fully interoperable with Java, which means that an application can use both Java and Kotlin classes, and they can interact with each other. Although Kotlin is not a new version of Java, it can be seen as an improvement when related to Android, since both are used to develop native Android applications. Overall, both support the same libraries (even if the library is in one language, since they are interoperable, both languages can use it). Some libraries are optimized for Kotlin.

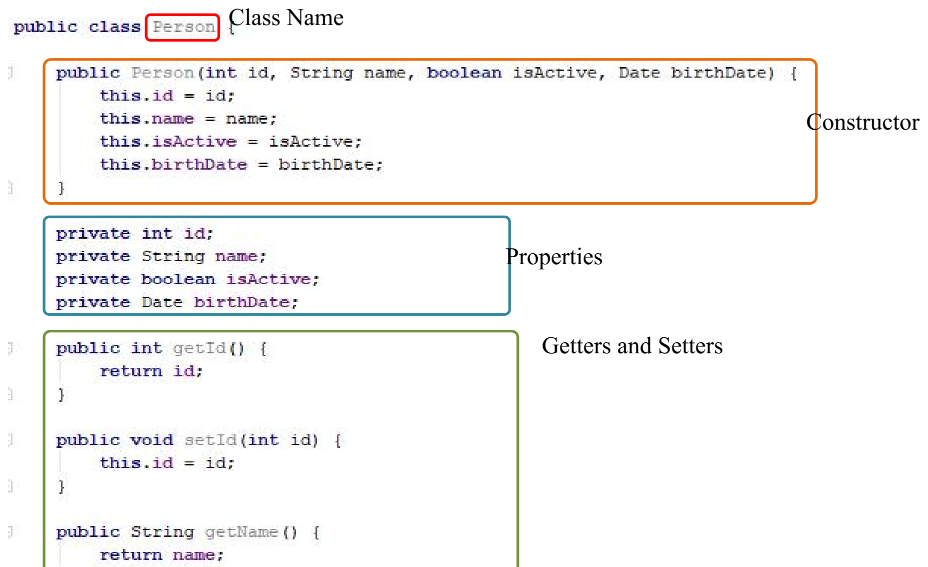
When deciding between Kotlin and Java, there are some points that the developers should take in question. Kotlin vs Java[67]

Java is a class based object-oriented programming language. Java has been for years the main language for Android development. It was not until 2017 that Google announced the support for Android app development with Kotlin. Even though Kotlin is the focus of a lot of attention lately, the Google Play (Android's app store) is still largely consistent of apps using Java [68].

One of the main differences between the two languages is the syntax. Java's syntax is very verbose, which means that its code could almost be read like a text. Although this makes the code very readable, it means a lot of writing to get something done, needing a lot of boiler-plate code (a lot of code to accomplish small functionalities).

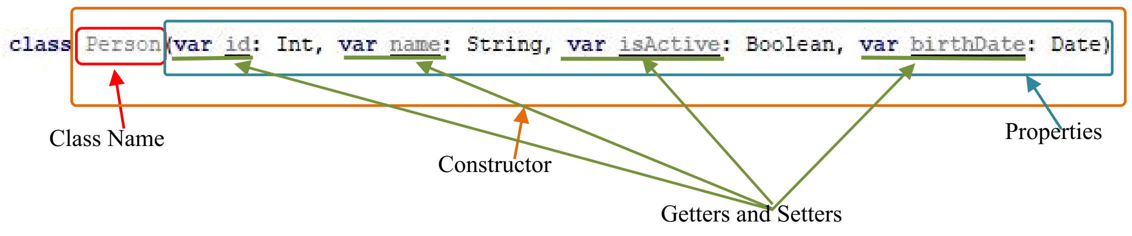
- Class declaration

In Figure 44, we have a Java class named Person, with four member properties, the constructor, and all their getters and setters. All this is necessary to manage this class. Although most of this can be generated via the IDE, it is still necessary to generate it.



```
public class Person {

    public Person(int id, String name, boolean isActive, Date birthDate) {
        this.id = id;
        this.name = name;
        this.isActive = isActive;
        this.birthDate = birthDate;
    }

    private int id;
    private String name;
    private boolean isActive;
    private Date birthDate;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
```

In Figure 45, we have the same class Person, but now written in Kotlin.



```
class Person(var id: Int, var name: String, var isActive: Boolean, var birthDate: Date)
```

Figure 45 - A class in Kotlin

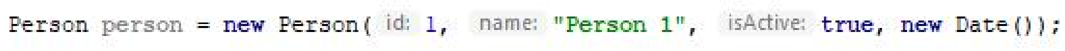
As we can see, the same class that took many lines to write in Java, takes only 1 line to write in Kotlin. In case we wanted fewer getters or setters, it would take less code to write the class in Java, but the class in Kotlin would not change much in terms of extension.

In Kotlin, the type of the variable comes after the variable’s name, which comes after the mutability of the variable. A “var” is a variable that can be reassigned, while a “val” is a read-only variable. Therefore, when we say a variable is a “var”, the compiler creates the getter and setter for it (because it can be reassigned). However, if we say it is a “val”, it only creates the getter (because it cannot be reassigned). If we want to make the variable private, just put “private” before the mutability identifier (example: “private var id: Int”). If we do not want a variable to be a member property of the class, do not put the mutability identifier (for example, “id: Int” without “var” or “val”); this way, the property will not be accessible outside the class’s constructor.

By writing that single line, we are creating a class Person with a single constructor that takes 4 arguments, with getters and setters for all its member properties.

- Initializing an object

In Java, to create an object of the type Person, we must write the code present in Figure 46

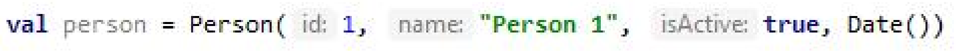


```
Person person = new Person( id: 1,  name: "Person 1",  isActive: true, new Date());
```

Figure 46 - Initialization of Person using Java

1. Define the type of the object: Person
2. Define the name of the object (person)
3. Call its constructor preceding with “new” with the arguments in the right order.

In Kotlin, we write the code present in Figure 45.



```
val person = Person( id: 1,  name: "Person 1",  isActive: true, Date())
```

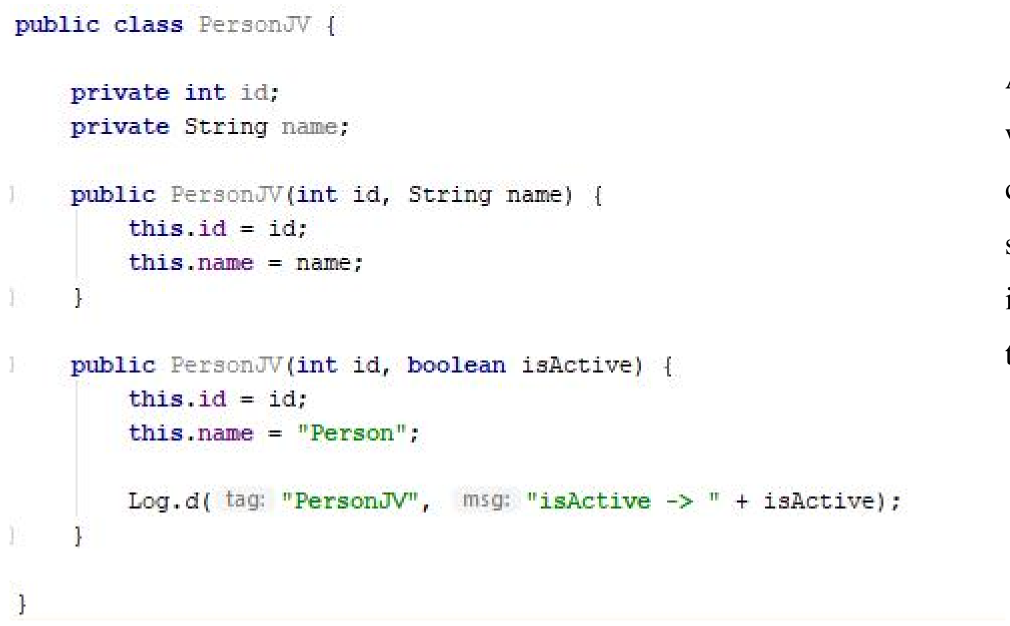
Figure 47 - Initialization of Person using Kotlin

1. Define the object’s mutability; in this case, it is “val”, which means it cannot be reassigned.
2. Define the name of the object (person).
3. Call its constructor; the arguments can be in the right order or not. If not, just say which argument it is by writing its name first (check birthDate and isActive). In Kotlin, there is no need to call “new” when creating a new object.
   Although we did not specifically say the type of the object, the compiler can infer its type by the assignment. If we wanted to specifically assign the type, write (in this case) “val person: Person = …”.

- Multiple constructors

Both Kotlin and Java allow multiple constructors.

In Java, constructors are declared the same way, as long as their signature do not match, this is, two constructors cannot have the same number of parameters with the same type in the same order. This is shown in Figure 48.

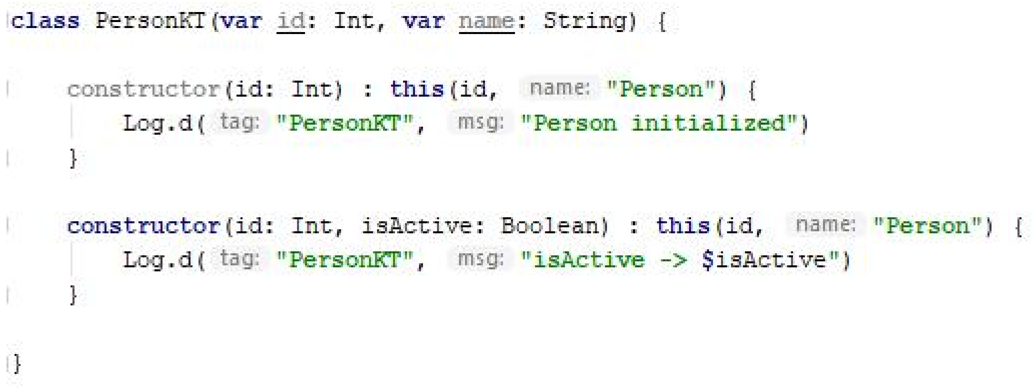


```
public class PersonJV {

    private int id;
    private String name;

    public PersonJV(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public PersonJV(int id, boolean isActive) {
        this.id = id;
        this.name = "Person";

        Log.d( tag: "PersonJV",  msg: "isActive -> " + isActive);
    }

}
```

Figure 48 - Constructors in a Java class

As we can see in the figure above, the first constructor initializes both member properties (id and name) with the inputs from the constructor, but the second constructor only does this with the id and initializes the name manually. We could not initialize any of the member properties, and they would hold the default value of its type.

In Kotlin, there are the concepts of primary constructor and secondary constructors. Each class can only have one primary constructor, but multiple secondary constructors. Like Java, two constructors cannot have the signature, this is, same parameters in the same order.



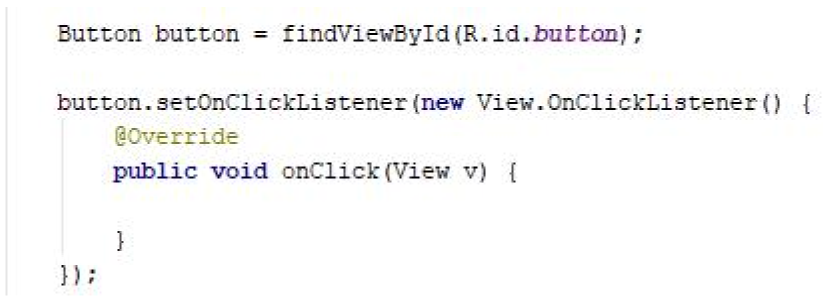
```
class PersonKT(var id: Int, var name: String) {

    constructor(id: Int) : this(id,  name: "Person") {
        Log.d( tag: "PersonKT",  msg: "Person initialized")
    }

    constructor(id: Int, isActive: Boolean) : this(id,  name: "Person") {
        Log.d( tag: "PersonKT",  msg: "isActive -> $isActive")
    }

}
```

Figure 49 - Constructors in a Kotlin class

As we can see in Figure 49, the first constructor (preceded by “PersonKT”) is the primary constructor. The constructors below are called secondary constructors. There can be as many secondary constructors as we want. Every secondary constructor must call the primary constructor (represented by “: this(…)”) because

- Accessing resources from the GUI layout

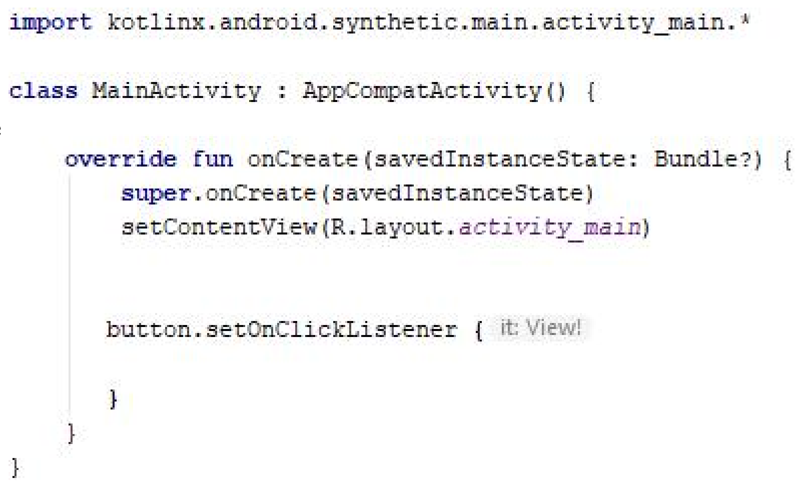
In Java, in order to access a resource from the GUI layout, we must first find it using “findViewById(id_of_object)”. If we are accessing the object multiple times, should put it in a variable to prevent searching it every time we want to use it, like in Figure 50.



```
Button button = findViewById(R.id.button);

button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {

    }
});
```

Figure 50 - Accessing a resource from the GUI layout using Java

In Kotlin, we can access GUI resources without needing to use the previously displayed method. By writing the id of the object, it will use the object on the respective view of the component we are using. For example, if we are in MainActivity and its view has a button with the id “button”, when we write “button” it will import the GUI resource and can be access as if it was found using findViewById(button). This can be seen in Figure 51. The first line of the image is importing all GUI resources (including the “button”).



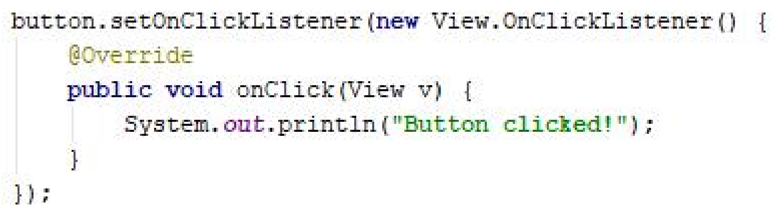
```
import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)


       button.setOnClickListener { it: View!

       }
    }
}
```

Figure 51 - Accessing a resource from the UI layout using Kotlin

The same way we access “button”, we can access every other GUI resource in this view, if they have an id to be accessed.

- Passing objects that implement interfaces

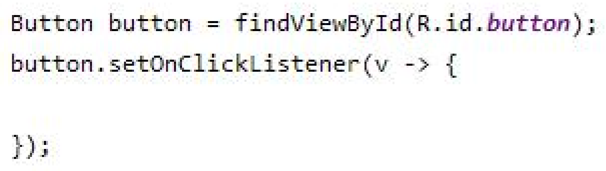
In Java, when we want to set, for example, a click listener in a button, we must use “button.setOnClickListener()” and pass a View.OnClickListener object. This object must implement an interface that has a “onClick” method that is fired when the button is clicked, as represented in Figure 52.



```
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        System.out.println("Button clicked!");
    }
});
```

Figure 52 - Catching a click event in Java

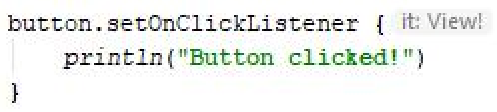
Later versions of Java offer a more compact way of doing this, shown in Figure 53.



```
Button button = findViewById(R.id.button);
button.setOnClickListener(v -> {

});
```

Figure 53- Catching a click event in later versions of Java

This way is much closer to the Kotlin’s way, present in Figure 54.



```
button.setOnClickListener { it: View!
    println("Button clicked!")
}
```

Figure 54- Catching a click event in Kotlin

These small differences make Kotlin’s syntax less readable (as in being harder to interpret, due to not being so verbose), but much cleaner, concise and faster to write.

Since Kotlin was created later, it was able to avoid some problems that Java has, and fix others. Some of the problems Java has that were addressed in Kotlin include:

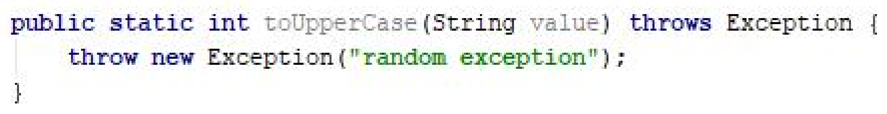
- Null references are controlled by the type system

All non-primitive Java types (represented with an uppercase first letter, e.g., “Integer”) are naturally nullable, which means they can hold a null value. Only primitive types (represented with a lowercase first letter, e.g. “int”) are unable to hold a null value. This can be a problem, since when accessing a method or property of a non-primitive object (for example, the length of a String), we must check previously that the object is not null. If we try to access a method or property of a null object, a NullPointerException is thrown, notifying the developer that he is accessing a null object, which is invalid. NullPointerException is one of the most thrown exceptions since it is easy to forget to check the value of an object previously. To solve this, Kotlin implemented nullable and non-nullable types.

In Kotlin, all the types start with an uppercase letter, even the primitive ones. However, even primitive types can be nullable (unlike in Java). To mark an object as nullable (whether it is primitive or not), simply declare it with a “?” after the type. For example, an object of type “Int” is non-nullable, however, an object of type “Int?” can hold a null value. When we access a nullable object’s properties or methods we must check if it is null or not. However, when we want to prevent the object from holding null values, just declare it without the “?”, and we never have to check its nullability, reducing the amount of validations necessary and prevent NullPointerExceptions. This means that in Kotlin, an object’s ability to hold a null reference is controlled by its type.

- No checked exceptions [69]

In Java, when performing an operation that can throw an exception, the exception must be caught, otherwise an error will be thrown notifying the developer that an exception was left un-handled. When a method can throw an exception, but does not handle it, the exception must be added to method’s signature, such as in Figure 55.



```
public static int toUpperCase(String value) throws Exception {
    throw new Exception("random exception");
}
```

Figure 55- Java method that throws an exception

Adding the exception to the method’s signature means that when we use that method, we must catch the exception, otherwise it will give an error, like in Figure 56.

```
String nameUppercase = UtilsJava.toUpperCase(name);
```

This would cause the application to be full of try-catches. According to *Effective Java*, exception should not be ignored [70]. However, it is said that in large projects, exceptions decrease productivity and have little to no increase in code quality [69].

To avoid this extra work, Kotlin does not required exceptions to be caught. However, there is the risk of an uncaught exception being thrown and the application crashing, so the developers should use them with caution.

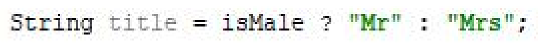
There are also some features Java has that Kotlin does not have, such has:

- Primitive types that are not classes [71]

Kotlin, like Java, has primitive types. However, even primitive types in Kotlin are classes (in Java, primitive types are not classes), which means that we can call member functions and properties on any variable. In Java, there are primitive types that are not classes, and, therefore, do not have member functions and properties.

- Ternary operator

The ternary operator is a simplified "if else"; it is composed by the condition, and the two possible results, the first, if the condition is true, and the second, if it is false. Although very compact, it can be sometimes hard to read, especially in more complex cases.

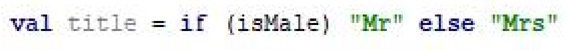


```
String title = isMale ? "Mr" : "Mrs";
```

Figure 57 - Ternary operator in Java

In Figure 57, the variable "title" is being set conditionally according to the value of a Boolean ("isMale"), representing whether the person is male or not.

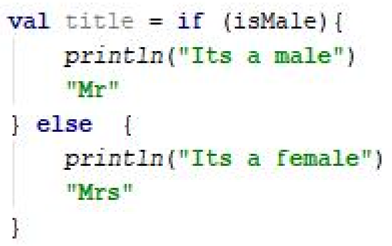
Kotlin does not have ternary operator, but a regular "if else" can be used to replace it. The "if" can return a value that will be assigned to the variable. Figure 58 represents Kotlin's version of the ternary operator.



```
val title = if (isMale) "Mr" else "Mrs"
```

Figure 58- Kotlin's equivalent of a ternary operator using "if else"

Since what matters is the value the if returns (the last variable of each branch, in this cases, “Mr” and “Mrs”), extra operations can be performed, as can be seen Figure 59.



```
val title = if (isMale){
    println("Its a male")
    "Mr"
} else  {
    println("Its a female")
    "Mrs"
}
```

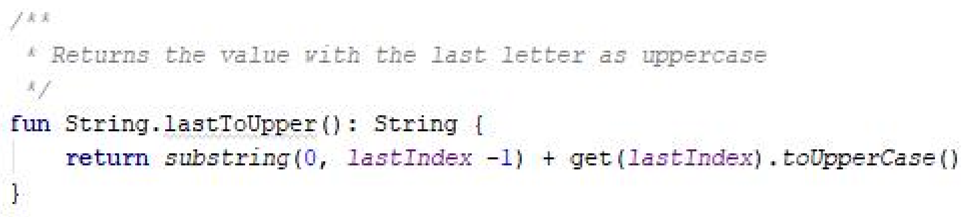
Figure 59- If else with extra operations

Kotlin also has some features that Java does not have; such has:

- Extension functions

Extensions consist in extending a certain type’s properties and members. Without changing the original class, we can add functionalities to it. This is particularly useful on classes that belong to the system or outside libraries.

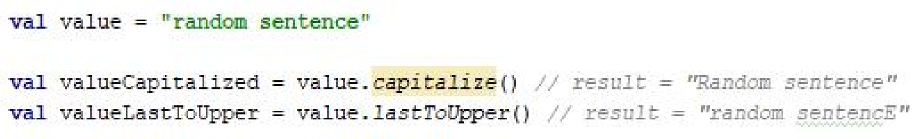
For example, we can add a function to String, to uppercase the last letter, as shown in Figure 60.



```
/**
 * Returns the value with the last letter as uppercase
 */
fun String.lastToUpper(): String {
    return substring(0, lastIndex -1) + get(lastIndex).toUpperCase()
}
```

Figure 60- An extension in Kotlin

Since String is a system’s object, we cannot change it. We can, however, add functions to it, called extensions. To use an extension, simply use it as if it was a normal String method.



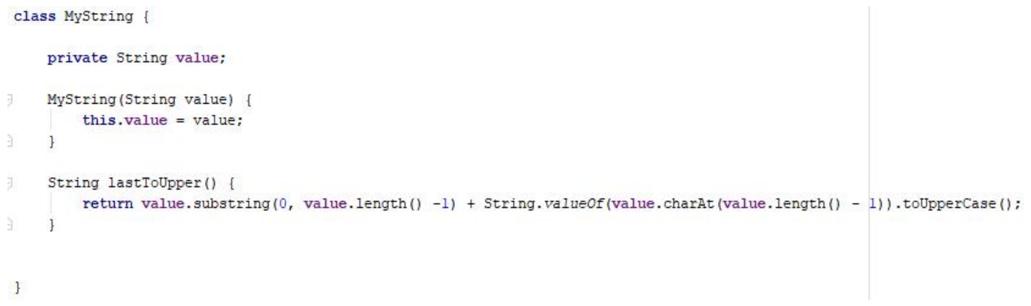
```
val value = "random sentence"

val valueCapitalized = value.capitalize() // result = "Random sentence"
val valueLastToUpper = value.lastToUpper() // result = "random sentencE"
```

In Figure 61, we have a String. We are using “capitalize()”, a String class function, and “lastToUpper()”, an extension created by us.

Even though Java does not have extensions, we can simulate it using the delegation pattern. Delegation consists in wrapping the class we want to “extend” in a class created by us and having an instance of said class in it, shown in Figure 62.

After that, just initialize a “MyString” object and use the methods as it if was a regular extension.

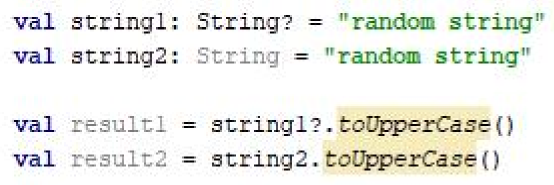
- Null safety



```java
class MyString {

    private String value;

    MyString(String value) {
        this.value = value;
    }

    String lastToUpper() {
        return value.substring(0, value.length() -1) + String.valueOf(value.charAt(value.length() - 1)).toUpperCase();
    }


}
```

Figure 62- Java delegation

Like it was said before, Kotlin has nullable and non-nullable types. When we want to access a member-property or function of a nullable-type object, it is possible that the object is null, causing a NullPointerException. To prevent this, Kotlin has a mechanism (called safe calls) that prevents the direct access to member-properties or functions of nullable objects. When accessing a member-property or function of a nullable object, we must put “?” before accessing the property of ı

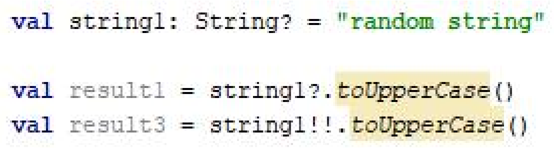


```kotlin
val string1: String? = "random string"
val string2: String = "random string"

val result1 = string1?.toUpperCase()
val result2 = string2.toUpperCase()
```

Figure 63- Safe call in Kotlin

Even though both “string1” and “string2” have values, string1 is marked as nullable (by the “String?”), so it’s possible that when accessing a function (in this case, “toUpperCase()”), its value is null. To avoid the need to previously verify if the value is not null, we can access it with a “?”, called a “safe call”. This means that if “string1” has a value, it calls the function normally, but if “string1” is null, it returns null. Since “toUpperCase()” returns a String, a safe call will return a nullable String.

It is also possible to make non-null asserted calls. Non-null asserted calls are marked by “!!” and can be interpreted as saying that the object is mandatory to be non-null, as shown in Figure 64.



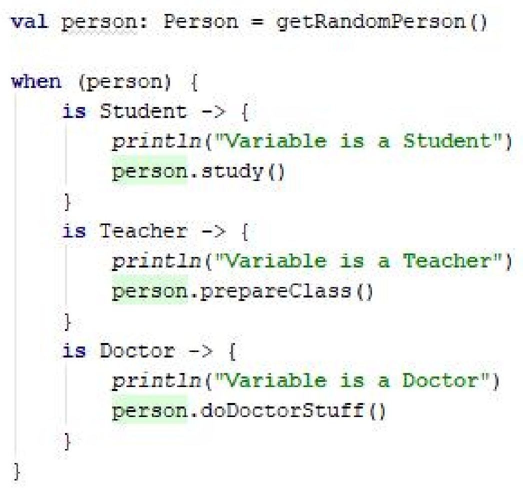
```
val string1: String? = "random string"

val result1 = string1?.toUpperCase()
val result3 = string1!!.toUpperCase()
```

Figure 64- Non-null assertion in Kotlin

However, if we access a member-property or function of a null object using a non-null asserted call, the application crashes, so caution is advised.

- Smart cast

In Kotlin, when we check that a variable is of a certain type, we can access it as if it were of that type, without needing to cast it. The same works for nullability check. This is called smart cast. In the figure above, “getRandomPerson()” returns an object that implements the interface “Person”. When we check its type, we can access the functions and member-properties of that type, shown in Figure 65. “study()” is specific of class “Student”, after we check that the object “person” is a Student, we can access the method.



```
val person: Person = getRandomPerson()

when (person) {
    is Student -> {
        println("Variable is a Student")
        person.study()
    }
    is Teacher -> {
        println("Variable is a Teacher")
        person.prepareClass()
    }
    is Doctor -> {
        println("Variable is a Doctor")
        person.doDoctorStuff()
    }
}
```

Figure 65- Smart cast in Kotlin

If a variable is of a nullable-type, when we check that it is not null (i.e. has a value different than null), we can access it as if it were of a non-nullable type.
This only works inside the validation (in this case, the “when”), not outside it.

- Lateinit

In Kotlin, we can mark a “var” as “lateinit” by putting “lateinit” before it (e.g. “lateinit var id: Int”). A lateinit variable is a variable that is not nullable but is not assigned at the moment of its declaration. This can be helpful on variables initialized through dependency injection or in the setup method of a unit test.

To check if a lateinit variable is initialized, we can use “::name_of_variable.isInitialized”, that returns a Boolean with true if it’s already initialized or false otherwise.

Appendix C - Functional Requirements – User Stories and Acceptance Criteria

- Login

  [Online] As a valid user, I want to be able to login on the application by providing my credentials. I must be associated with a veterinarian or a veterinarian's assistant, and to at least one brigade.

  Acceptance criteria: As a user, I know the login is successful when I am presented with entities and brigades to enter.
- Enter entity/brigade

  [Online] As a validated user, I want to be able to enter a brigade to which I, as a veterinarian or veterinarian's assistant, am associated to.

  Acceptance criteria: I know that I have entered a brigade once I am able to start registering data in the application and consult data regarding that entity and brigade.

  [Offline] As a validated user, I want to be able to enter a brigade of an entity in which I previously entered.

  Acceptance criteria: Same as online, but only with brigades in entities previously entered.
- Seed the local database

  [Online] When I enter a brigade, I want to have the data of its entity seeded into my local database, so I can use it while offline.

  Acceptance criteria: I know that the data was successfully seeded when I can perform work in the application.
- Logout

  [Offline/Online] As an authenticated user, I want to be able to logout of the application.

  Acceptance criteria: I know that I have successfully logged out once I am presented with the login screen and I must provide my credentials to login.
- Change entity/brigade

  [Online] As an authenticated user, I want to be able to change the current entity/brigade in which I am working on, to another brigade that I am associated with, without having to logout.

  Acceptance criteria: I know I have successfully changed entity/brigade when I am once again able to register data in the application,

- Holding CRUD

  List holdings: As an authenticated user with permission to consult holdings, I want to be able to list the holdings of the entity in which I am authenticated in, seeing their unique identifier, name, and active state. I also want to be able to consult the holdings' details, seeing its unique identifier, name, short name, tax number, active state, default aptitude, responsible brigade, and corresponding country zone.

  Acceptance criteria: I should see all the fields listed above present in each holding. I also want to be notified when no holdings are available to list.

  Update holdings: As an authenticated user with permissions to update holdings, I want to be able to update a holding, changing the fields listed above, except the responsible brigade.

  Acceptance criteria: I know that I have successfully updated a holding when the holding's record shows updated with the new data.

  Create holdings: As an authenticated user, I want to be able to create a new holding, filling the fields listed above, except the responsible brigade, which should be locked as the brigade in which I am working on.

  Acceptance criteria: I know that I have successfully created a holding when it can be found in the list of holdings.

- Owner/Keeper CRUD

  List owners/keepers: As an authenticated user with permissions to consult owners/keepers, I want to be able to list the owners/keepers of a given holding that belongs to the entity in which I am authenticated in, seeing their unique identifier, occupational unique identifier, name and active state. I also want to be able to check their details, seeing their unique identifier, occupational unique identifier, title, name, birth date, tax number, active state, whether they are a company (if it's an owner), and if they are not a company, their gender (always shows gender if it's a keeper). I also want to be notified when no owners/keepers are available to be listed.

  Acceptance criteria: I should see the fields listed above, with the proper labels.

  Update owners/keepers: As an authenticated user with permissions to update owners/keepers, I want to be able to update an owner/keeper, changing the fields listed above.

  Acceptance criteria: I know that I have successfully updated an owner/keeper when the owner's/keeper's record shows the updated data.

Create owners/keepers: As an authenticated user, I want to be able to create a new owner/keeper, filling the fields listed above. The owner/keeper should be automatically assigned to the holding previously selected.
Acceptance criteria: I know that I have created an owner when the owner is listed in the owners of the premises previously selected.

- Animal CRUD

  As an authenticated user, I want to be able to list the animals of any holding that belongs to the entity in which I am working on, seeing their preferred identification, group, and species. I also want to be able to check their details, seeing the

    - group
    - species (must belong to the selected group)
    - breed (must belong to the selected species)
    - aptitude
    - identifications (all the configured in the WebApp; only the visual and preferred are required)
    - preferred identification (one of the filled identifications)
    - gender
    - birth date
    - mother (must be of the same species, required according to the species' configuration)
    - father (must be of the same species, required according to the species' configuration)
    - premises
    - owner (must be working on the selected premises)
    - keeper (must be working on the selected premises)
    - color (optional)
    - status
    - name (optional)

  Acceptance criteria: When listing, I should see the preferred identification of the animal, along with its group and species. I also want to be notified when no animals are available to be listed. When checking its details, I want to see the fields listed above.

As an authenticated user, I want to be able to register new animals, filling the fields listed above, making it available to be intervened.

Acceptance criteria: I know that I have successfully created an animal when it is visible in the list of animals of that holding and is available to be intervened.

As an authenticated user, I want to be able to update an animal so that its information in the system matches the reality. I must be able to update the fields listed above, except the identifications (which require a re-identification, that is a different operation), group, species and current premises (which require a movement, created in a different operation).

Acceptance criteria: I know that I have successfully updated an animal when its details show the updated data.

- Service

  Create service: As an authenticated user, I want to be able to register interventions in animals, by applying actions to them.

  To start a service, I must first specify the setup data:

  - the holding in which I am working
  - the group of animals to which I am intervening
  - the date of the intervention

  The brigade responsible should be the one I am working on.

  If a previously created service matches this data, I should not be able to create another one with that data.

  Acceptance criteria: I know that I have successfully started a service when I am able to select the default actions for the animals that will be intervened. If I the setup data matches a previously created service, I should be warned that I cannot create another one. If the setup data matches a non-synchronized service, I should be given the option to continue it.

  Define template actions: After starting the service, I should be able to select the default actions to be applied to facilitate the intervention of animals. I should not be able to progress without selecting at least one action, since no animal can be in the service without actions. I should be able to change the default actions at any moment without affecting the previously intervened animals of that service.

Acceptance criteria: I know that I have successfully defined the default actions when I am able to select the animals to be intervened, and their actions are the same ones as the default actions.

List animals intervened: I should be able to list the animals being intervened in a service, seeing their preferred identification and the unique identifier of the actions being applied to them. I should also be able to check and update the actions being applied to them.

Acceptance criteria: The animals being listed here should be the ones where actions have been applied to in the current service.

List animals not intervened: I should be able to list the animals that are yet to be intervened in the service, seeing their preferred identification, group, and species. The animals should be of the selected holding and group. When no animals are available to be intervened, I should be notified.

Acceptance criteria: The animals mentioned above should be the ones of the selected group belonging to the selected holding that are yet to be intervened.

Create non-intervened animals in service: I should be able to create animals in the context of the service without intervening them, filling their fields, so they can be available to be intervened. Animals created while in a service should be deleted if the service is deleted.

Acceptance criteria: When creating an animal, the holding and group should be locked to the previously selected holding and group. When the animal is created, it should be visible in the list of non-intervened animals of that service.

Update non-intervened animals in service: I should be able to update animal not being intervened while in the service. If I do not have permission to update them, I should only be able to check their details.

Acceptance criteria: I know that I have updated an animal if their details show the updated data.

Intervene animals manually: I should be able to list the animals of the previously selected holding so that I can intervene the animals manually by selecting them and confirming their intervention.

Acceptance criteria: The animals list should be the animals of the holding previously selected that are yet to be intervened in this service. Once I select the animals and confirm their intervention, they should no longer be visible in this list, and should be present in the list of animals intervened on the service.

Read identifications with RFID readers: I should be able to intervene an animal by reading its RFID tag with an RFID reader connected to the application.

If when reading the identification:

- the animal is yet to be intervened, but is present in the list of non-intervened animals of the service, it should be intervened
- the animal is already intervened, I should be redirected to its details, seeing the actions that are being applied to it
- the animal does not exist in the holding, I should be asked if I want to create it, filling the fields of the normal creation of the animal (except the group that should be locked to the group of the service, and the electronic identification that should be locked to the identification read) and selecting the actions being applied to it. After creating it, it should be visible in the list of intervened animals of the service.

Acceptance criteria: If the animal was not intervened, it should no longer be present in the list of non-intervened animals and should be present in the list of animals intervened in the service. If the animal was created, it should be present in the list of animals intervened in the service. If the animal was already intervened, it should be in the same list as it already was.

Change the actions being applied to an animal: I should be able to change the actions that are being applied to an animal after it already is in the service.

Acceptance criteria: I know that I have successfully changed the actions being applied to an animal if those actions are displayed correctly in the list of animals intervened under the correspondent animal.

Updating an animal being intervened: I should be able to update an animal being intervened, editing its details as if it were a regular update outside the service.

Acceptance criteria: I know that the animal was updated when its details show the updated data.

Delete service : As an authenticated user, I want to be able to delete a non-synchronized service, deleting all the actions applied to animals in it and the animals created while performing the service.
Acceptance criteria: I know that I have successfully deleted a non-synchronized service when it no longer shows in the list of services performed.

- Summaries
  View services made in current session: I must be able to list previously made services made by my brigade on holdings of the entity in which I am working on, seeing the date of the service and the corresponding animal group. If the service is yet to be synchronized, I must be able to change it, otherwise, I must only be able to check its contents without changing the animals or the actions being applied to them.
  Acceptance criteria: I must see the services my brigade has done in the current session, group by holding, showing the date the service was made and the group of animals intervened in the service.
- Devices Library:
  List available devices: I should be able to list available Bluetooth, BLE, and Wi-Fi devices to connect to them, showing their MAC address and their name.
  Acceptance criteria: Devices that have been made available and are in range should be displayed. Each device should also display its status (not connected, connecting, and connected). If no devices are available, I must be notified.
  Connect with multiple devices: I should be able to connect to certain devices via Bluetooth, Bluetooth Low Energy (BLE) or Wi-Fi so they can interact with the application, such as RFID stick readers or printers.
  Acceptance criteria: I know that the device was successfully connected when it can interact with the application.
- Synchronization
  Work while offline: As a user, I want to be able to work while offline, so that I can work in areas where internet connection is not possible.

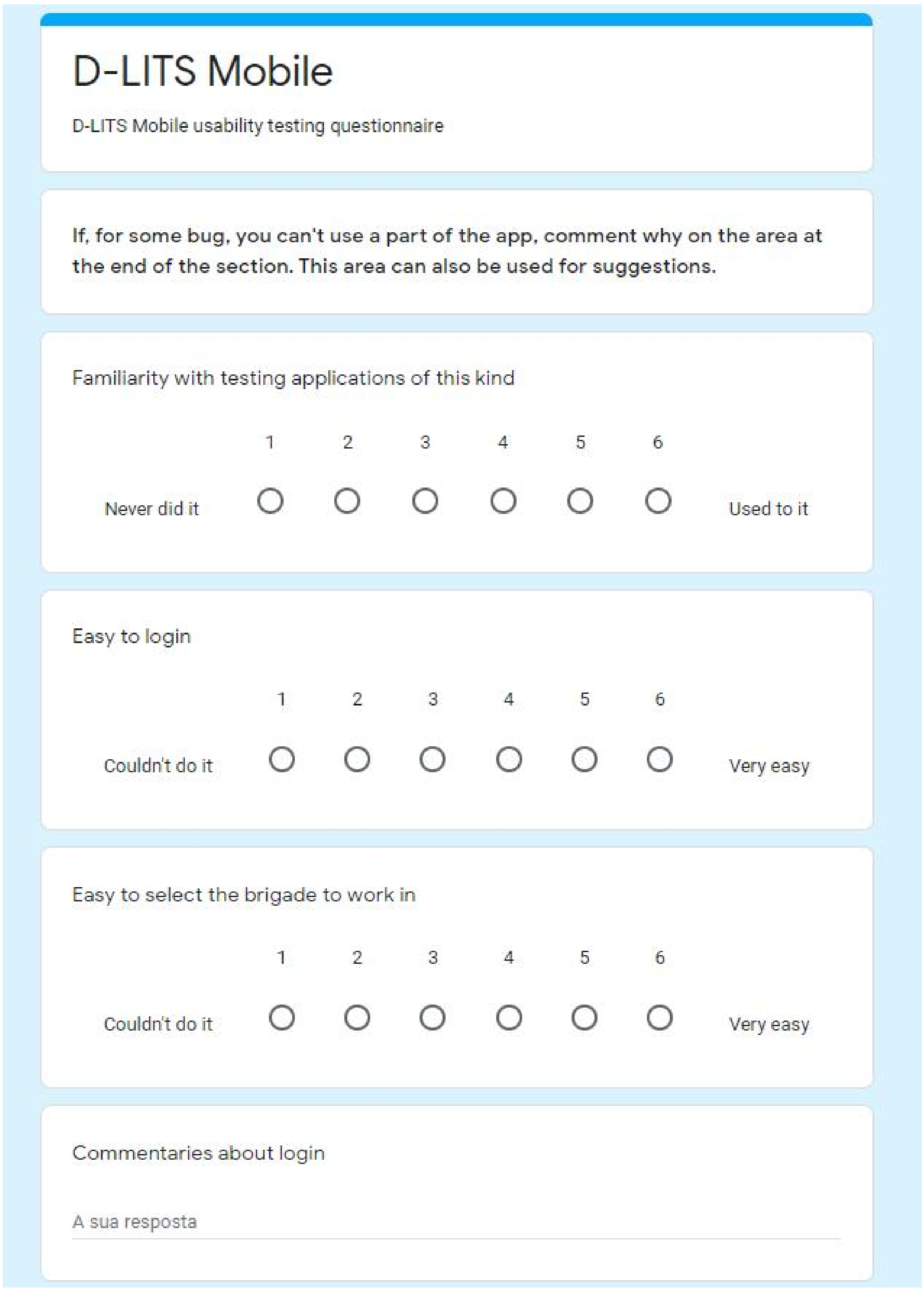
Acceptance criteria: While offline, I must be able to perform most of the operations supported by the application, excluding login, and entering an entity not previously entered (entering an entity that I already entered while online is possible). All my work should be saved.

Automatic synchronization: As a user with connection to internet, I want to have my work performed while offline automatically (without my intervention) synchronized with the remote API, so that all the work is saved and made available to other users. I also want to have relevant data changes in the remote API to be synchronized to my application, so that I can use it.

Acceptance criteria: The work I performed should be synchronized without my interaction. The data should be synchronized in a way to avoid data inconsistency. I know that the changes made by other users were successfully synchronized to my application when the data that is displayed to me changes. When the data is successfully synchronized, I must be notified.

Appendix D – Usability Tests Questionnaire

In this appendix, we can check the usability questionnaires provided for users for usability tests.



D-LITS Mobile

D-LITS Mobile usability testing questionnaire

If, for some bug, you can't use a part of the app, comment why on the area at the end of the section. This area can also be used for suggestions.

Familiarity with testing applications of this kind

	1	2	3	4	5	6	
Never did it	○	○	○	○	○	○	Used to it

Easy to login

	1	2	3	4	5	6	
Couldn't do it	○	○	○	○	○	○	Very easy

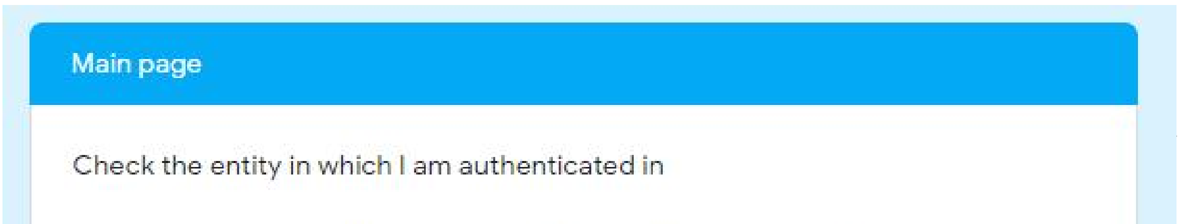
Easy to select the brigade to work in

	1	2	3	4	5	6	
Couldn't do it	○	○	○	○	○	○	Very easy

Commentaries about login

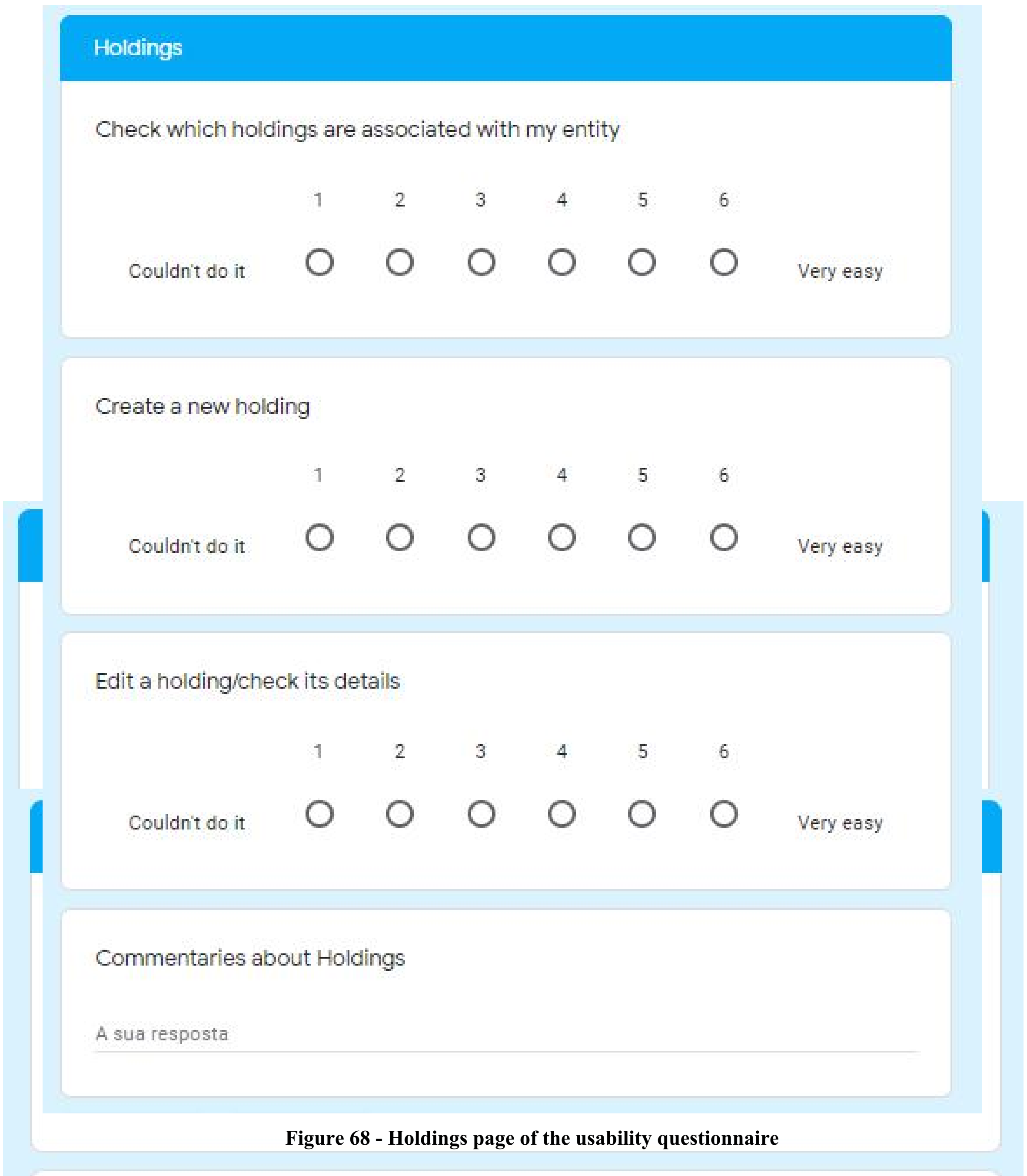
A sua resposta

Figure 66 – First page of the usability questionnaire



Main page

Check the entity in which I am authenticated in



Holdings

Check which holdings are associated with my entity

	1	2	3	4	5	6	
Couldn't do it	○	○	○	○	○	○	Very easy

Create a new holding

	1	2	3	4	5	6	
Couldn't do it	○	○	○	○	○	○	Very easy

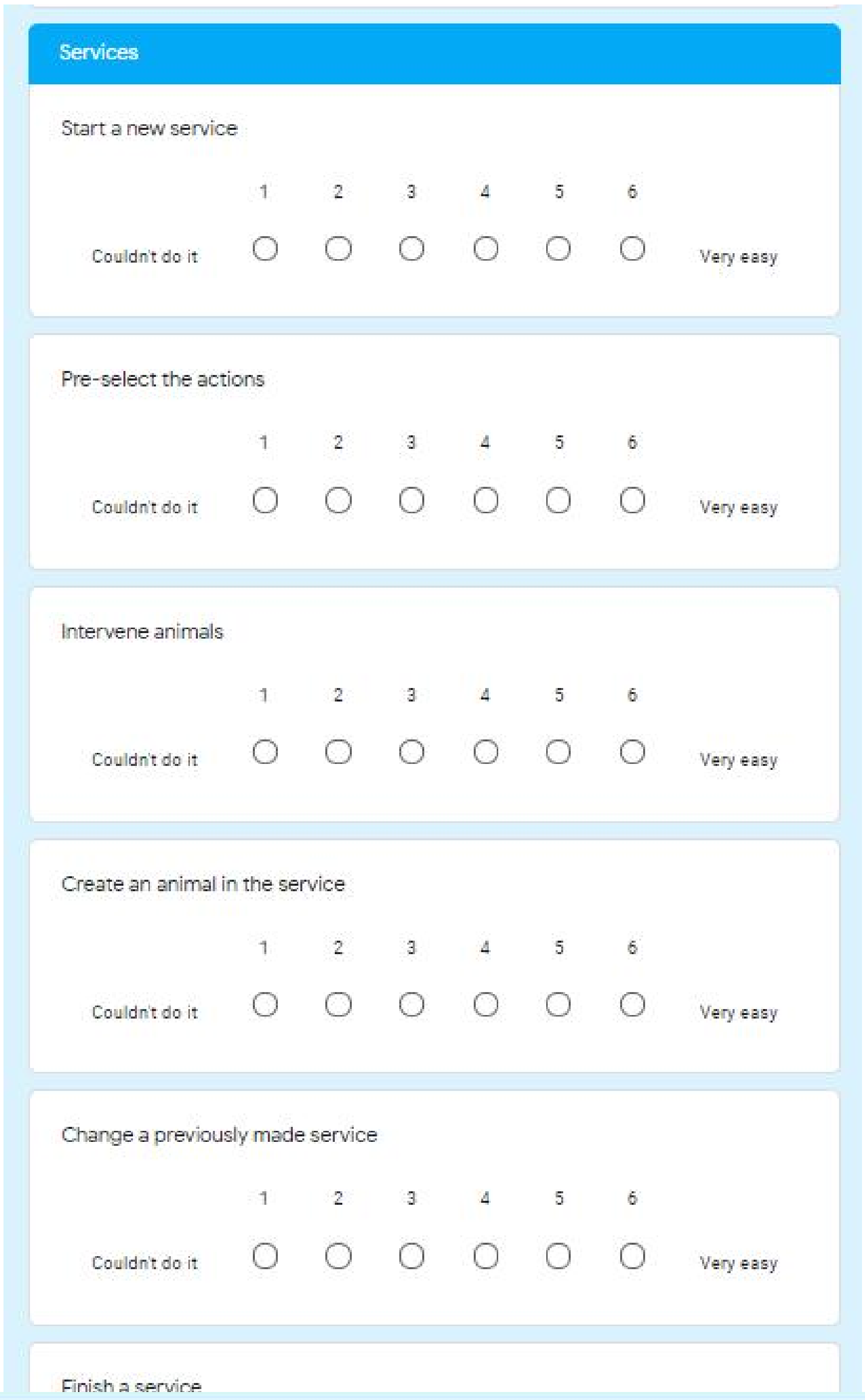
Edit a holding/check its details

	1	2	3	4	5	6	
Couldn't do it	○	○	○	○	○	○	Very easy

Commentaries about Holdings

A sua resposta

Figure 68 - Holdings page of the usability questionnaire



Services

Start a new service

	1	2	3	4	5	6	
Couldn't do it	○	○	○	○	○	○	Very easy

Pre-select the actions

	1	2	3	4	5	6	
Couldn't do it	○	○	○	○	○	○	Very easy

Intervene animals

	1	2	3	4	5	6	
Couldn't do it	○	○	○	○	○	○	Very easy

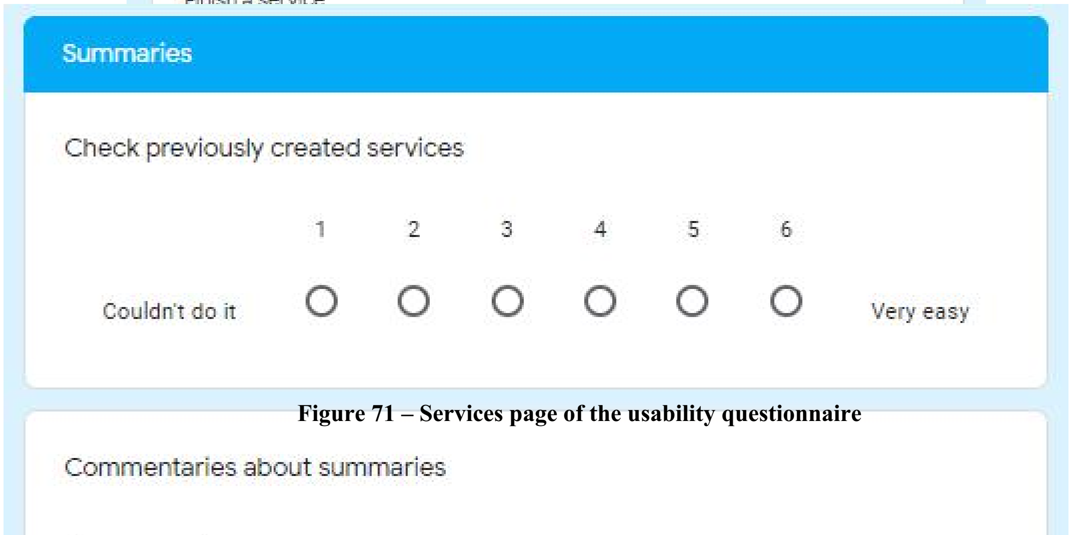
Create an animal in the service

	1	2	3	4	5	6	
Couldn't do it	○	○	○	○	○	○	Very easy

Change a previously made service

	1	2	3	4	5	6	
Couldn't do it	○	○	○	○	○	○	Very easy

Finish a service



Summaries

Check previously created services

	1	2	3	4	5	6	
Couldn't do it	○	○	○	○	○	○	Very easy

Commentaries about summaries

Figure 71 – Services page of the usability questionnaire