Building a mobile application for a recycling startup: Oust!

Bachelor Thesis

Sylvain Losey

August 2019

Thesis supervisors:

Prof. Dr. Jacques Pasquier-Rocha
and
Pascal Gremaud
Software Engineering Group

Software Engineering Group
Department of Informatics
University of Fribourg (Switzerland)

Acknowledgments

I would like to express my gratitude to Prof. Dr. Jacques Pasquier and Pascal Gremaud for their guidance during this project. I also want to thank the Faculty's Exams Delegate, Prof. Dr. Thierry Madiès, for allowing me to write this thesis.

I want to give a special thanks to my friends and colleagues of *Oust!* for their enthusiasm throughout this adventure.

Finally, I am very grateful to my family for their continuous support and encouragement.

Abstract

Recycling is a vital tool to prevent climate change and use the planet's resources efficiently. Unfortunately, it can be inconvenient or inaccessible to some people.

In this report, we will present *Oust!*, a Swiss startup that aims to simplify recycling. To help in this task, we will build a mobile application with Flutter, allowing us to create an Android and iOS version from a single codebase.

Keywords: Mobile application, Flutter, Redux, Django, REST, Recycling

Table of Contents

List of Figures

List of Tables

Listings

1 Introduction

Our society advances at an increasing pace, and it seems that our standards of living improve with each passing year. Unfortunately, we are also starting to realize that this progress has consequences.

1.4 million students recently protested in the school climate strikes [6] to prevent further global warming. In January 2018, China banned the import of recyclables from foreign countries [7], which highlights the weaknesses in our global waste management strategy.

Thoughtful use of Earth's finite resources and preservation of our climate are going to be two significant challenges of the 21st century. Recycling is one of the ways to address these challenges, but it can sometimes be a burden. In this report, we will introduce an alternative way of recycling and see how technology can help us in this domain.

1.1. Motivation and goals

This report will be centered around *Oust!*, a startup that aims to simplify recycling. *Oust!* offers at-home recycling through a subscription, as well as one time pickups of objects and recyclables to discard.

The company was founded in 2017 by four students of the University of Fribourg. We knew each other previously and launched it outside of the context of University, but we had the chance to improve some aspects of the company through written works and projects. This includes this bachelor thesis in which we will explore the existing software architecture of the company, and build a new mobile application for customers to access our services easily.

The application will be built with Flutter, allowing us to create an Android and iOS app with a single codebase. The goal is to have an app ready to be used by the end of this report. It should allow customers to book and manage our services as seamlessly as possible.

1.2. Outline

Chapter 1: Introduction
The introduction provides an overview of the contents of this report

Chapter 2: Context
First, will see how recycling systems work in general, and how *Oust!* fits in the larger picture of waste management. We will also define in more details what services we offer, and why we decided to build a mobile app for our customers.

Chapter 3: Global Architecture
In this chapter, we will describe the complete software architecture of *Oust!*. Most of the backend was developed before this report, but a REST API will be implemented for the app we are building.

Chapter 4: Design process
We will then discuss the design process of the app. In the first sections, we will see how the technology we used was chosen, and then the steps we took to build the UI and UX of the app. In the last section of this chapter, we will present the interface that was implemented and explain how it works.

Chapter 5: Development
This chapter presents how the app was built. First, we review the tools we used with some examples. Then we will explore in more details the most challenging areas of the implementation.

Chapter 6: Conclusion
Finally, the last chapter discuss the results of this report and offers a personal perspective on the project as a whole.

2
Context

2.1. Waste management

In many countries, increasing the recycling rate has been defined as an important political objective. However, recycling has a cost, and it can vary widely depending on the collection system used.

2.1.1. Recycling systems

Single, dual and multi-stream recycling

We can separate recycling systems into three types, depending on the number of material streams. In single-stream recycling, everything goes in a single bin, which is usually left on the curbside and picked up by a truck. The recyclables are then separated at a Materials Recovery Facility (MRF) through manual and automated sorting. This method has been growing in popularity in North America in recent years [1]. A variant of this system is dual-stream recycling, which works similarly but instead of one bin, paper fibers such as newspapers and cardboard boxes are sorted in a second bin. A comprehensive study [2] found that dual-stream recycling had a lower overall cost than single-stream. Higher collection costs were offset by lower processing costs and higher resale value due to the better quality of the resulting material. The third type of recycling system is multi-stream recycling, where the depositor sorts recyclables into multiple types of materials. There is usually no MRF involved. This system is more common in Europe, and especially in Switzerland where citizens are asked to bring their recyclables to collection centers accepting up to 50 different types of recyclables [8].

Bringing and hauling systems

A second distinction between the different systems is whether citizens are in charge of the transport to a facility, or if another entity is responsible for it. Whether a bringing or hauling system is used often depends on the number of recycling streams. In single and dual-stream recycling, materials are usually hauled from the curbside by a truck. For multi-stream recycling, a common method is collection centers where citizens bring their recyclables. Hauling of specific materials is, however, also organized in some regions, usually more urban ones.

Systems used in different parts of the world

No single system has imposed itself throughout the world, which shows that the best option is not self-evident. The fact that, in general, North American countries moved from multi-stream recycling to dual or single-stream recycling indicates that they were considered superior or at least more convenient. However, this may have been predicated on the assumption that China would continue to import a large part of the resulting recyclables, which have a high contamination rate relative to multi-stream recycling. China banning imports of recyclables with a contamination standard higher than 0.5% changes the situation. The economics of recycling worldwide have been altered as a consequence of this ban, resulting in many municipalities resolving to send waste to landfills instead of recycling it [10].

2.1.2. Situation in Switzerland

In Switzerland, where our company is active, a bringing multi-stream recycling system is the most common. Federal law requires cantons to establish a waste management plan[1]. Cantons, in turn, give the responsibility of waste management to municipalities, usually by requiring collection centers to be established[2]. This means most municipalities have collection centers available, which citizens regularly visit to deposit their recyclables. Smaller collection points accepting the most common recyclables and designed to be at walking distance are also used. Larger municipalities sometimes offer to haul some recyclables such as paper and cardboard on the curbside.

Collection through recycling centers, with recyclables separated from the beginning of the valorization process, yields materials of high quality [9]. It is also relatively cost-effective for municipalities because there are no direct collection costs for recyclables, in contrast to single or dual-stream recycling that is generally implemented with hauling. This cost, however, does not disappear. It is reported on citizens that have to use their time and usually their vehicles to transport recyclables. This can be a burden as collection centers may have restrictive opening hours, requiring citizens to visit them during their leisure time.

[1] Environmental Protection Act (EPA) Article 31 Paragraph 1 - `https://www.admin.ch/opc/en/classified-compilation/19830267/index.html#a31` - Accessed 31/07/2019

[2] Example of the Canton of Fribourg: Loi sur la gestion des déchets (LGD) Article 14 - `https://bdlf.fr.ch/app/fr/texts_of_law/810.2` - Accessed 31/07/2019

Alternatives

Elderly people, in particular, have a hard time accessing recycling centers. They often do not have a vehicle or have difficulty lifting heavy loads. As a result, they must find other solutions such as asking family members to do it, but this is not always possible. Sadly, we have found that a large number of them have to dispose of their recyclables in the trash.

Some other options have been developed to help with this problem. Mr Green, for example, is a company active in the German-speaking part of Switzerland that proposes to its customers to put all their recyclables in a plastic bag, unsorted. The bags are then picked up and sorted by them. This has the benefit of solving the inconvenience of transport but introduces the problems of single-stream recycling: the resulting recyclables can be of lower quality, and processing costs are high, especially in this case as everything is sorted manually.

2.2. Oust!

2.2.1. The idea behind *Oust!*

Many people have a strong desire to recycle, but the task can be time consuming and inconvenient. When we started the company, we realized the shortcomings of the different systems and thought there could be an alternative. As a result, we had the idea to offer a subscription service to take care of this burden. Customers would just have to leave their sorted recyclables at their front door, or wherever is most convenient, and we would pick it up at regular intervals. Effectively, we would offer a hauling multi-stream recycling system. We started very small, by distributing flyers and picking up recyclables with our private cars at first. We quickly saw that there was a strong demand and bought utility vehicles more adapted to the task.

This system has the advantage of being more efficient than collection centers, since one vehicle can pickup up the recyclables of around 15 customers in one round, resulting in much fewer kilometers traveled than if the 15 took their car to go to the recycling center. We can see an example of this in Figures 2.1 and 2.2, which are a graphical representation of one of our rounds of 30 customers. In Figure 2.1, we see the path our customers would take if each of them took their car to go to the recycling center. Figure 2.2 shows the path one of our vehicles takes to collect recyclables, consisting of two rounds of 15 customers.

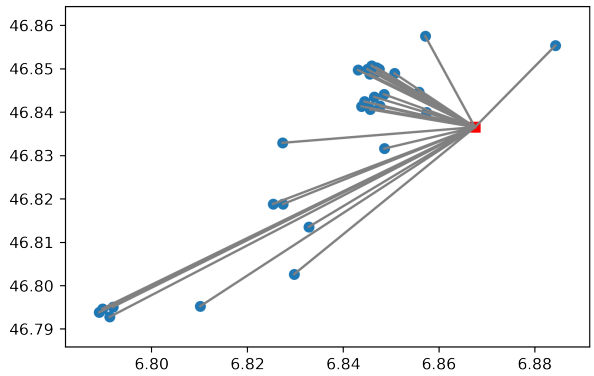

Fig. 2.1.: Route used by customers

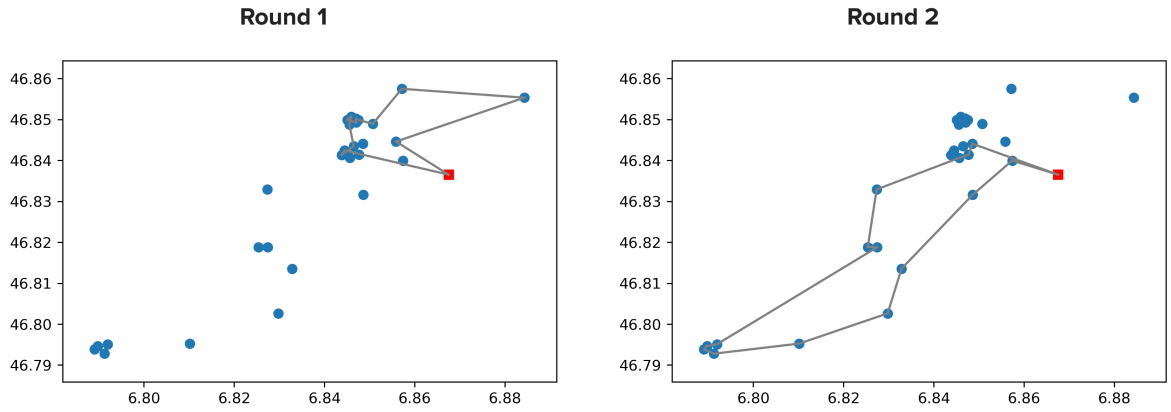

Fig. 2.2.: Route used by *Oust!*

The more customers we have, the less distance between them, making it more efficient and ecological since less gas is used for transport. From an economic standpoint, we can accomplish the task of transporting recyclables in much less time and with fewer resources than if the customer was doing it himself. Table 2.1 shows the results of the two variants illustrated in Figures 2.1 and 2.2. For the customer route model, these results come from the sum of the driving distance and duration from each customer to the recycling center. For the *Oust!* route, they are the sum of the same distance and duration values between customers in the order shown in Figure 2.2. This route is optimized to get the fastest path visiting each customer. For both models, the values for the distance and duration between points come from the Open Source Routing Machine (OSRM), which we will discuss later.

	Customers route (Figure 2.1)	*Oust!* route (Figure 2.2)
Total distance	510.27 km	128.41 km
Total duration	6:40:10	2:05:36

Tab. 2.1.: Comparison of customers and *Oust!* rounds

As can be seen in Table 2.1, the optimized route with a utility vehicle requires less than a third of both time and kilometers traveled than the route customers would take. Thanks to this increased efficiency, we can offer competitive prices profitably if we have enough customers in a given area.

2.2.2. Next steps

Up until now, we have not had a real customer web or mobile application. Most of the subscriptions are requested through phone calls or registration on the website, which is a simple email form, followed by an appointment at the home of the customer.

As our customer base grew, we have been asked more and more to do one-time pickups of a sofa or a closet, for example, and to bring them to a recycling center. One of the problems we faced when customers booked what we call lifts was the difficulty of giving a quote with only text information. A closet can be tiny and picked up in two minutes, or very large and requiring multiple employees to transport it. In most cases, pictures of the materials to dispose of were the most effective way to communicate and allowed us to give an accurate price. However, it is not very practical to ask for pictures by email for each new request. As a result, the idea of a mobile app became very appealing, as it would allow taking pictures of the different materials and get an accurate quote in a few steps. It would also let customers get a subscription directly from the app without an appointment and manage it easily.

3
Global Architecture

3.1. Overview

The software system we use features two main parts, depicted in Figure 3.1. A backend stores all relevant data and performs the necessary processes. Multiple front ends let customers and employees interact with the data stored in the backend. In the middle, a REST API allows the two parts to communicate.

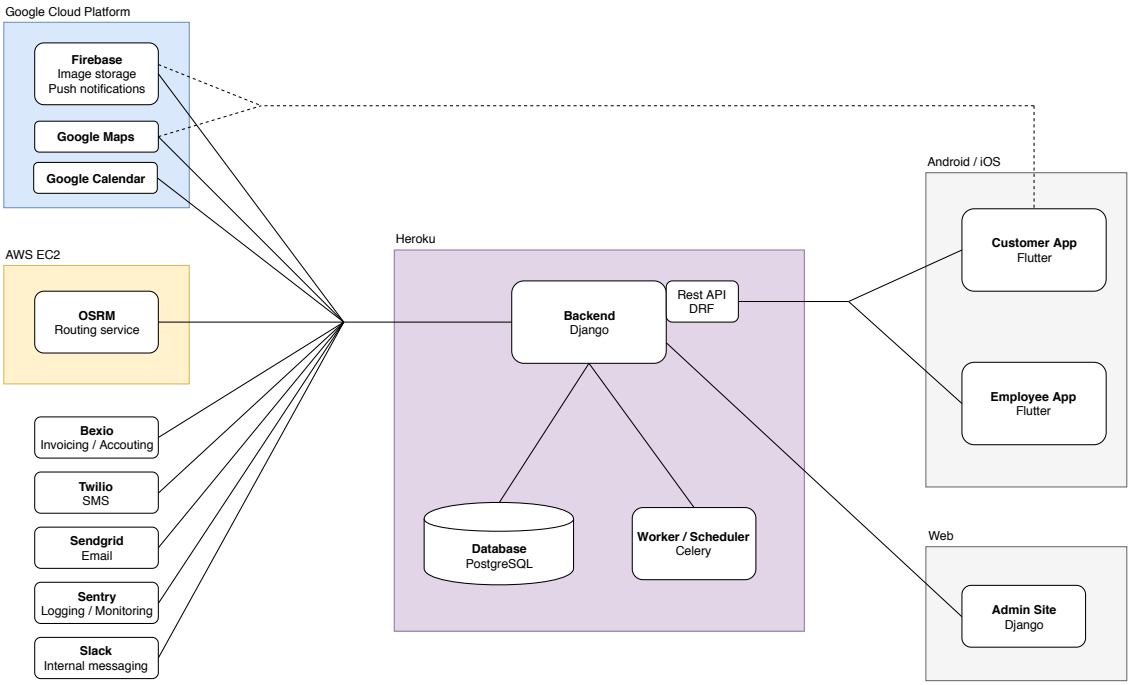

Fig. 3.1.: Software architecture of *Oust!*

3.2. Backend - Django

Different technologies were considered to build the backend, and Django, a Python-based framework was chosen for several reasons. The framework's slogan is "The Web framework for perfectionists with deadlines" [11], which is an excellent fit for a startup that usually wants to build and use a product fast.

3.2.1. Advantages of Django

- Learning the basics of the framework and the Python language is considered relatively easy
- Highly customizable admin interface to manage resources stored in the database
- The data we need to store is structured, making a relational database better suited than a NoSQL database. Django is built primarily for relational databases.
- The main plugin used to create REST APIs for this framework, Django Rest Framework, is very well made and easy to use.

3.2.2. Technologies used

Several packages and external services were used to accomplish a variety of different tasks.

Celery

Celery is an asynchronous task/job queue based on distributed message passing [12]. Using such software was necessary in our case as Django does not support asynchronous tasks. Celery is handy to perform work that is not urgent, such as updating data in external APIs. Coupled with Celery Beat, a plugin for Celery, it allows us to schedule repeating tasks following a CRON schedule. We use this to automate many business processes, such as sending payment reminders.

Bexio

Bexio is a Swiss company that offers invoicing and accounting software through a subscription [14]. We chose this solution as most of the data on their platform is accessible through an API, which allows us to automatically create invoices and synchronize their payment status with our database, for example. We found that it was also important to use software specifically designed for the country served as some features change between regions and are usually not customizable, such as generating the correct payment slips or following the accounting rules of the country of operation.

Firebase

Firebase is a platform developed by Google which offers many services to help with creating web and mobile applications [16]. It was chosen for its ease of implementation, and very comprehensive integration with Flutter, the tool we use to create mobile applications, which we will see in more detail in Chapter 5. We use Firebase to store images sent by customers as it is generally not recommended to store them in a database. The link of the image is then stored in our database. We also use the push notification service, which allows us to send notifications to both Android and iOS users.

Google Calendar

Our first idea to manage appointments with customers was to create our own scheduling server, but we quickly realized that this could be a very time consuming and complicated task. A frontend to display and allow CRUD operations on appointments would also have to be implemented. As employees already use Google Calendar to add appointments manually, we decided to use Google's API and store scheduling information directly in Google Calendar [17]. The API is flexible and easy to use, allows users to create as many calendars as needed and most importantly, to keep using Google Calendar for inputting appointments scheduled via phone calls, for example.

Open Source Routing Machine (OSRM)

As we serve customers at their physical location, we have a lot of geographical data and operations to perform. At first, we used Google Maps and its API but were quickly limited by its cost, and by constraints imposed on some operations. Retrieving a matrix of distance and travel duration between customers, for example, is limited to around 12 customers, but our rounds can go up to 40 customers per vehicle. Therefore, we turned

to a free open source project, OSRM, a routing engine offering most of the functions we needed [18]. There is a limited demo test server running OSRM, but to be used in production OSRM is usually self-hosted. We chose to deploy it on an Amazon Web Services EC2 instance (t2.small) as it is one of the most popular options for the task, and is relatively cheap [19]. This allows us to have our own routing engine to generate optimal subscription start dates, for example.

3.2.3. Hosting - Heroku

The backend is hosted on Heroku, a *Platform as a Service* company, which makes building, running and scaling applications very easy [20]. It supports many languages including Python, and once implemented it reduces the deployment process to a single terminal command. Heroku may be more expensive than a simple server or instance on other hosting providers, but the time gained in managing the deployment and the security it brings makes it well worth it in our case. Our application runs on 3 *Hobby* dynos. One is used to host the Django gunicorn server, one for the Celery worker, and one for Celery Beat, which schedules tasks to send to Celery.

Heroku also provides equally easy to install and manage add-ons for various tools and services. In our case, we use a Heroku Postgress add-on as a database, and a Redis cloud message broker for Celery. Dynos and add-ons are easily scalable as the application grows. For example, our Redis cloud usage fits into the free plan, but for the Heroku Postgres database, we recently switched to a paid plan with more capacity.

3.3. REST API - Django Rest Framework

Different pieces of software need to access data stored in the backend.

- Employee Android/iOS mobile application used to provide all necessary information when completing rounds and recording the time necessary to complete each customer
- Customer Android/iOS mobile application that allows customers to create and manage subscriptions, as well as order one-time lifts.
- A custom software we use to optimize rounds and complete them as efficiently as possible. It will be hosted on an AWS EC2 instance.

Other projects are planned for the following months, such as rebuilding the customer-facing website which will retrieve and post live data to the backend and might run a web version of the customer mobile app, when Flutter fully supports it.

A widely used method to allow all of these parts to communicate is RESTful APIs. Representational State Transfer (REST) is an architecture style described by Roy Fielding for designing loosely coupled applications over HTTP [3]. REST defines a set of constraints but does not give rules regarding implementation.

3.3.1. Definition of REST

Six architectural constraints define a true RESTful API:

Client-server
The goal of this constraint is to enforce separation of concerns between the server and clients

Stateless
The server does not keep track of any state data between sessions. The clients send all necessary state information with each request.

Cacheability
The server should define if its response is cacheable or not, allowing the client to reuse the response for equivalent requests if the data is cacheable.

Layered system
REST allows using a layered architecture where the APIs are deployed on a server, the data stored in another and the authentication handled on a third server for example.

Uniform interface
This is the main distinguishing factor between REST and other styles. The idea is to decouple implementation from the services provided, to allow server and clients to evolve independently.

Code on demand
This constraint is optional and was not implemented in our API.

3.3.2. HATEOAS

The uniform interface constraint defines four subconstraints, of which our API follows three. Hypermedia as the engine of application state (HATEOAS), the fourth one, specifies that a REST client should return links to related resources instead of URIs. This would allow someone to discover all resources of the API by following the links, without needing prior knowledge of the structure of the data.
We chose not to implement this, even though it would be easy to do using DRF. Our client applications are relatively small, and the API is intended to be consumed by internal developers who have a good knowledge of the data structure. Furthermore, using HATEAOS would complicate the logic on the frontend without bringing clear benefits in our case.
As a result, we can consider the API implemented a RESTful API, but not a truly REST API. Following Leonard Richardson's levels of REST [21], the API developed is at level 2, the final level 3 being a REST interface using HATEOAS.

3.3.3. Implementation

One of the reasons for choosing Django as our framework was the ability to use Django Rest Framework, making it easy to develop our API.

We use the Django ORM (Object Relational Mapping) with models that contain data and methods for each type of resource. We can see an example of the Phone Number model in Listing 3.1.

```
class PhoneNumber(models.Model):
    customer = models.ForeignKey('Customer', on_delete=models.CASCADE)
    phone_number = PhoneNumberField()
    number_type = models.CharField(choices=PHONE_NUMBERS_TYPES)
    reminder = models.BooleanField()
```

List. 3.1: Example of a Model in Django

Setting up a standard endpoint with GET, POST, PUT, PATCH, and DELETE operations is very straightforward. First, we define a serializer for the model, as shown in Listing 3.2.

```
class PhoneNumberSerializer(serializers.ModelSerializer):
    class Meta:
        model = PhoneNumber
        fields = '__all__'
```

List. 3.2: Example of a Model Serializer in DRF

DRF can serialize and deserialize most standard data types by default, and, if necessary, defining custom logic for a field is simple. We then implement a viewset that takes two required arguments: a serializer and a queryset, which is the query to the data exposed by the endpoint. There are optional arguments, here we also define a permission class, allowing access to the resource if it is linked to the current user or if the user is an admin.

```
class PhoneNumberViewSet(viewsets.ModelViewSet):
    serializer_class = PhoneNumberSerializer
    queryset = PhoneNumber.objects.all()
    permission_classes = [IsCustomerOrAdmin]
```

List. 3.3: Example of a Model Viewset in DRF

Result

Django REST framework automatically creates a browsable API when the requested format is HTML, such as when the endpoints are accessed through a web browser. Figure 3.2 shows an instance of a phone number viewed with admin credentials. We found this functionality very useful, and it made developing the API more convenient.

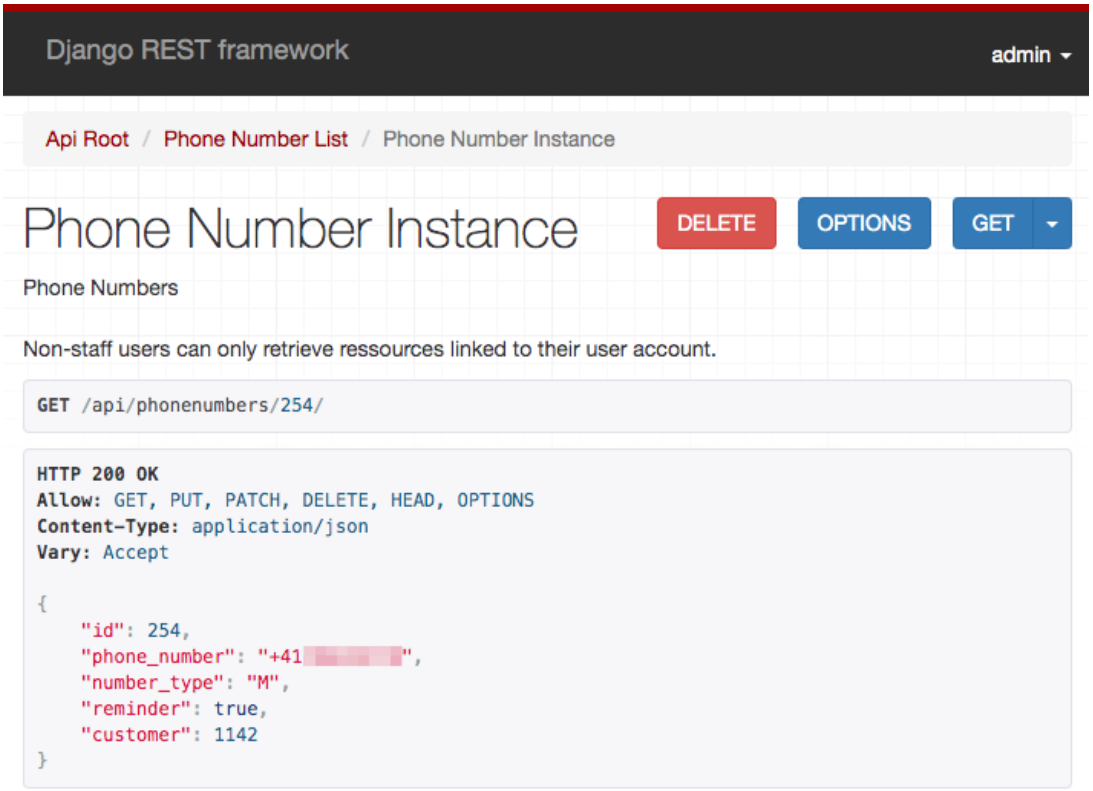

Fig. 3.2.: Example of an instance viewed on the browsable API

3.3.4. Swagger

Having documentation of each endpoint of an API can be very handy, and we chose Swagger for this task. Swagger UI creates an interactive HTML webpage displaying each endpoint with its properties and allows direct interaction with the API [22]. To implement it, we use a Django package called drf_yasg (Yet Another Swagger Generator) [23]. This package auto-generates OpenAPI 2.0 schemas from the Django REST Framework code, which are then displayed by Swagger. The Swagger documentation for the API we built is available here: `https://admin.oust.ch/api/swagger/` at the time of writing, however, most of the endpoints are not public.

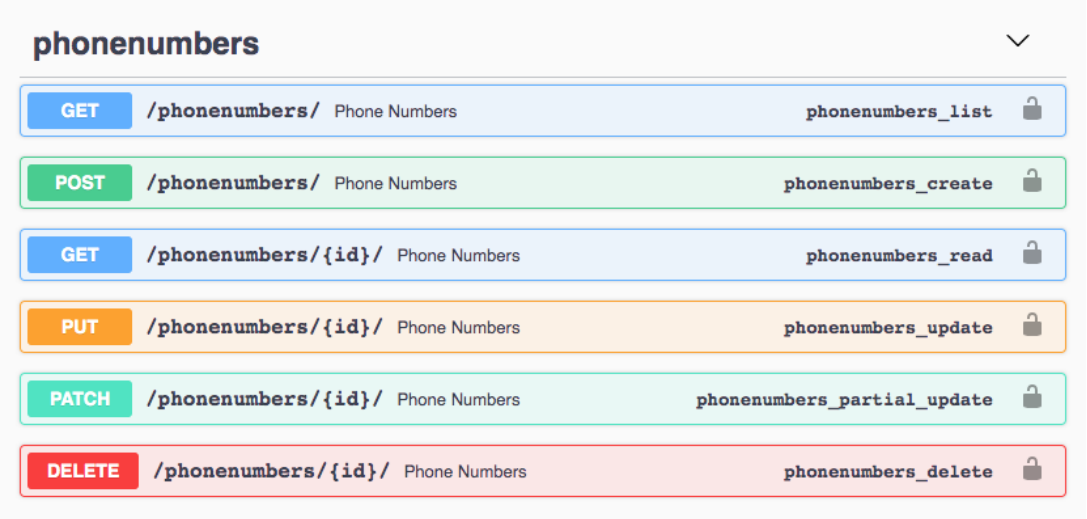

Fig. 3.3.: Example of a model's endpoints viewed on Swagger

Filtering

Several options are available when building an API to filter the results or find related resources. We decided to use query parameters in the URL, which allows us to filter queries with several different parameters concurrently. For example, if we want to request the phone numbers associated with a specific customer, customerId 1002 let's say, we can add the query parameter `/?customerId=1002` to the end of the request. We could add other filters if necessary, such as type of phone number, and use them simultaneously.

3.4. Frontend

We have not included our web frontend `www.oust.ch` in Figure 3.1, as it is a static website with no connection to the backend for now.

Currently, the other existing front ends are not customer-facing. We customized the built-in admin interface of Django to manage everything related to the backend from the admin side. We also have an internal mobile application used to help drivers complete rounds.

The center of our attention, however, is going to be the customer mobile application that was developed for this report. The next two chapters will go into detail about the design and development of this app.

4
Design process

4.1. Overview

To get the most of our limited development resources, we tried to incorporate UI (User Interface) and UX (User Experience) best practices from the beginning. As a result, we followed this plan:

- Gather requirements and select the most appropriate technologies to use
- Build a prototype of the app and continuously user test it
- Once we have a good prototype of the UI, develop the app itself

In this chapter, we will go through those steps and discuss the main challenges encountered with each one.

4.2. Requirements gathering

The tool we choose to build the app should satisfy the following requirements:

- Good performance on both iOS and Android
- High development velocity and easy to learn
- Support for Google Maps
- Support for push notifications
- Ability to register credit cards

4.2.1. Available technologies

Several tools can be used to build mobile apps, which can be divided into two main groups.

Native development

The first one is building the app in the designated language of the operating system, usually Java or Kotlin for Android, and Objective-C and Swift for iOS. This method is used by most large companies as the resulting app usually has the best performance and uses the native components of the operating system. However, this requires two separate apps written in different languages, essentially doubling the work required. For smaller companies, this can be very hard to do, and might not be the most efficient way to operate.

Cross-platform development

Ideally, one would have a single codebase that can be deployed on several platforms. This is the promise of cross-platform development tools, but this can come with significant drawbacks. Some of the popular tools are React Native, Xamarin, PhoneGap, and Ionic. Each has a slightly different way of operating, which, unfortunately, usually has some major disadvantage. React native, for example, is not fully cross-platform and frequently requires some parts to be written in native iOS and Android. The performance usually is not on par with native apps, resulting in large companies that were using React Native, such as Airbnb, deciding to turn away from it [24].

Flutter

One of the newest cross-platform development tools, Flutter, was initially released by Google in 2017, with its 1.0 stable release coming out in December 2018 [25]. Flutter takes a radically new approach by completely sidestepping the native platforms. The framework could theoretically run anywhere, as it draws every pixel on the screen itself through Google's Skia Graphics Engine, which also powers Chrome. It currently supports Android and iOS. Desktop and web versions are available in technical preview.

Flutter has several advantages over its competitors. Hot reload increases development speed as changes to the code are reflected instantly, in stark contrast to the long compile times of other tools. It usually has excellent performance in tests because Flutter is directly compiled to native code and executed very efficiently. The rise in popularity in the short time it has been available makes apparent the warm welcome that the developer community has given it. It was ranked third most loved framework in the Stackoverflow 2019 Developer Survey [26].

4.2.2. Tool used for our app

Given the good fit with our requirements and the features currently in development such as web support, we decided to use Flutter to build our mobile app. It has a plugin for Google Maps developed by the Flutter team itself, and support for push notifications through Firebase, which is built by Google too and is very well integrated with Flutter. Flutter only falls short on the last requirement: the ability to easily register credit cards, as no official plugin exists for the major credit card payment providers such as Stripe. However, this is not a deal-breaker since this functionality will be implemented later, and we expect better options to be available by then.

4.3. App design

The top priority we set for our mobile app was to be as simple to use as possible. A large part of our target audience is not necessarily used to booking services through a mobile app. If we want the app to be successful, using it should be intuitive, and getting a subscription or booking a lift would have to be seamless. That is not easy to do in general, and especially in our case, where we need quite a few data points to perform the tasks mentioned above.

To achieve that goal, we tried to inform ourselves as best as possible about the current best practices. One of the ways to do this was to follow online courses, such as the ones on *learnux.io* [27]. We also did some research with books, most notably *The design of Everyday Things* by Don Norman which is seen by many as one of the best books about design basics, very useful to know for UI/UX design [4]. We used the *10 Usability Heuristics for User Interface Design* as guidelines throughout the process [5] [28].

The general idea when designing a mobile application is to start from the problems the app is trying to solve. With that in mind, the goal is to get from a vague idea to a reasonably high fidelity prototype that a developer can implement. These are the steps we followed:

1. **Problems discovery**: User and market research
2. **UX Design**: User journey maps, sketches of major components.
3. **Prototyping**: Wireframes, low fidelity prototypes.
4. **UI Design**: Mockups of final UI, high fidelity prototypes.

4.3.1. Problem discovery

As we started the company around two years ago, we have a good idea of the problems we want to solve for our customers, and of the most practical ways to do it. A common practice in this phase is to perform user interviews. We were able to skip this step as we start each subscription by a meeting with the customer, which is a good opportunity to learn more about them and what they need. We also have a good idea of our target audience because of these appointments.

Before starting the company, we thought that the majority of our customers would be elderly people, but we found that around 70% of them are younger than 50 years old. As mobile apps generally see a better adoption in young people, this was a motivating factor in deciding to build an app. In general, we have a good sense of our customer's needs, and therefore did not feel the need to go through the step of establishing Personas or "Jobs To Be Done" forms.

4.3.2. UX Design

User Experience is how a user experiences and interacts with a product. Our goal here is to design the app to maximize user satisfaction when using it. This means making it simple, logical, and efficient to use, but also taking into account how the average user feels when using the product. We had to choose how to structure the app, and what the best method to get from a new user to a subscribed customer was.

For the structure, we quickly decided on using two separate bottom tabs, one for lifts and one for subscriptions. A third bottom tab would be the profile with general customer information, invoices, and payment methods. We think that the most appropriate way for a user to get a subscription or lift is through a wizard form: multiple forms separated into several steps that are often easier to go through than single page forms. The idea is that the form asks for the information it needs dynamically. If a user does not have an account yet, one will be created while completing the wizard form, whereas an authenticated user will not see an account creation screen, for example. A user that wants either a subscription or lift should just have to follow the steps of the form, and after the last step, everything should be ready for him.

At this point, we have a general idea of how the app would work. The next step is to think of the concrete steps a user would take through the app. This generally takes the form of a user flow chart as we can see in Figure 4.1. Not every step is represented, but the major ones should be there.

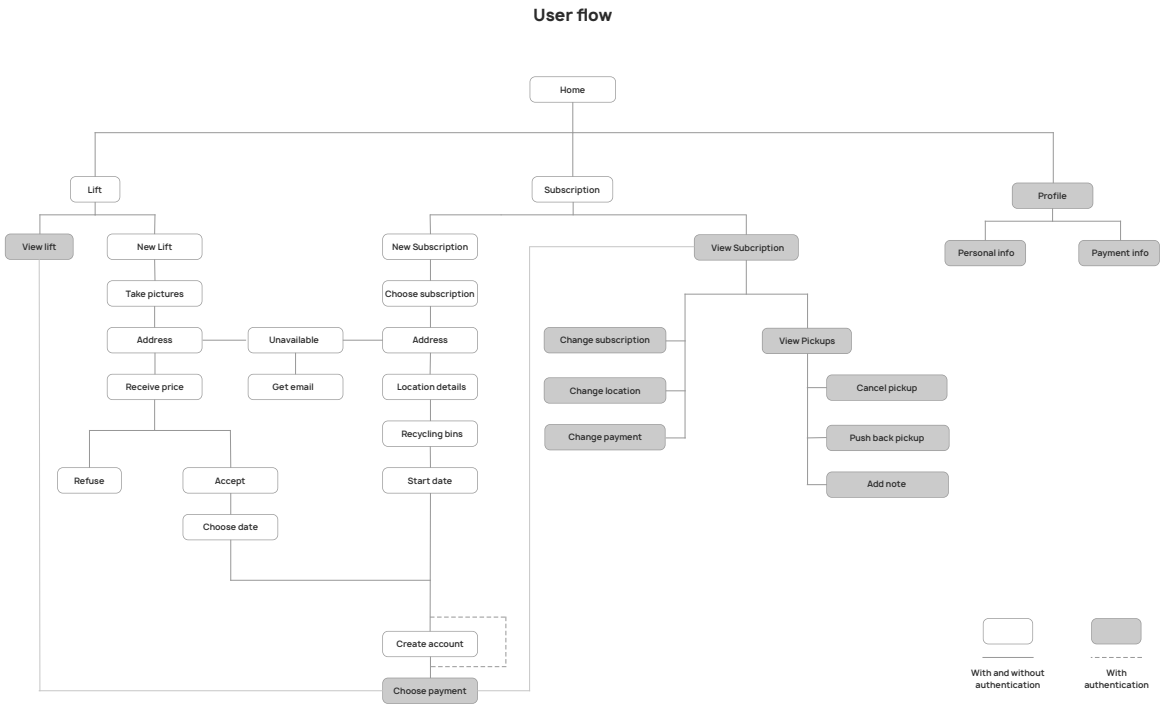

Fig. 4.1.: User Flow of the app

We want a user to be able to explore as much of the app as possible before having to create an account. We know that if the first screen we show when opening the app is a sign in or sign up screen, we will lose a large part of users right away. This way, the user can discover what the app is about before committing to creating an account. This should help get a higher retention rate during the signup process.

4.3.3. Prototyping

Now that the general structure of the app is set, we get to the largest part of the UI design process: deciding how each screen will work. Wireframes are usually used here. They are similar to blueprints, as they do not go into details about the appearance. The goal is to arrange different elements, such as buttons, titles, lists, or text fields, to determine how screens will function. Many tools can be used to draw wireframes, but we started with hand-drawn sketches for the early steps. When we had a good idea of the major screens, we switched to Sketch, a vector graphic software specifically designed for UI and UX design [29]. In Figure 4.2, we can see an example of the wireframe of some screens from the profile tab designed in Sketch.

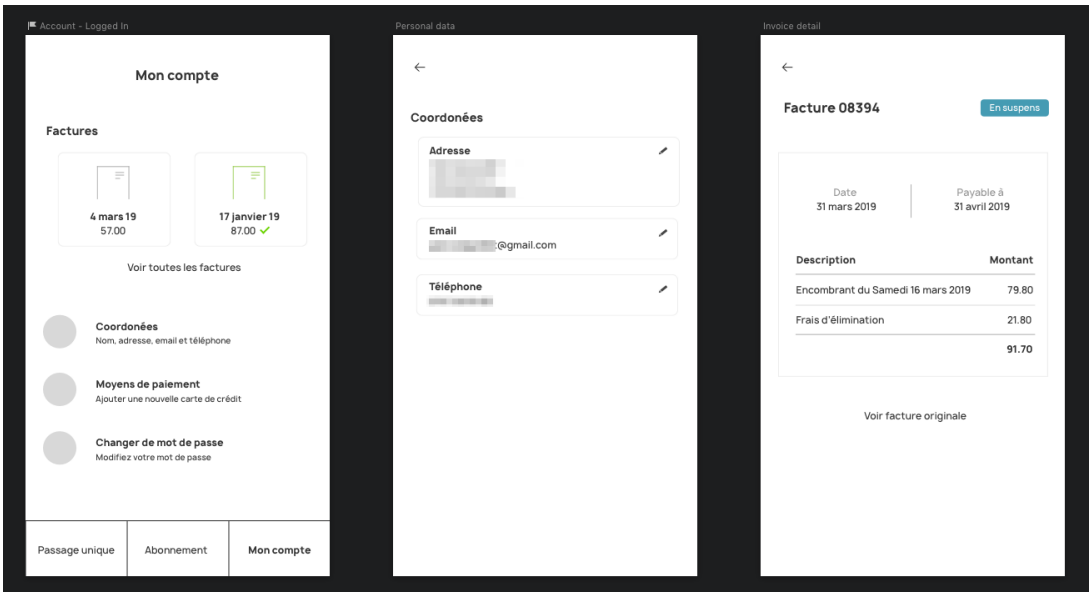

Fig. 4.2.: Wireframe in Sketch

Sketch has a prototype function that links one part of a screen, a button, for example, to the screen that should appear when the button is clicked. This results in a low fidelity prototype available online [30]. We used this prototype to test our ideas with friends and family, updating it with the feedback received continuously.

4.3.4. UI Design

The last step of the design process is to define the appearance of the app . Due to time constraints, we could not give this step as much time as we would have wanted to. We designed some screens, as shown in Figure 4.3, to get a general idea of what the final design would be, but it is more an indication than a final design.

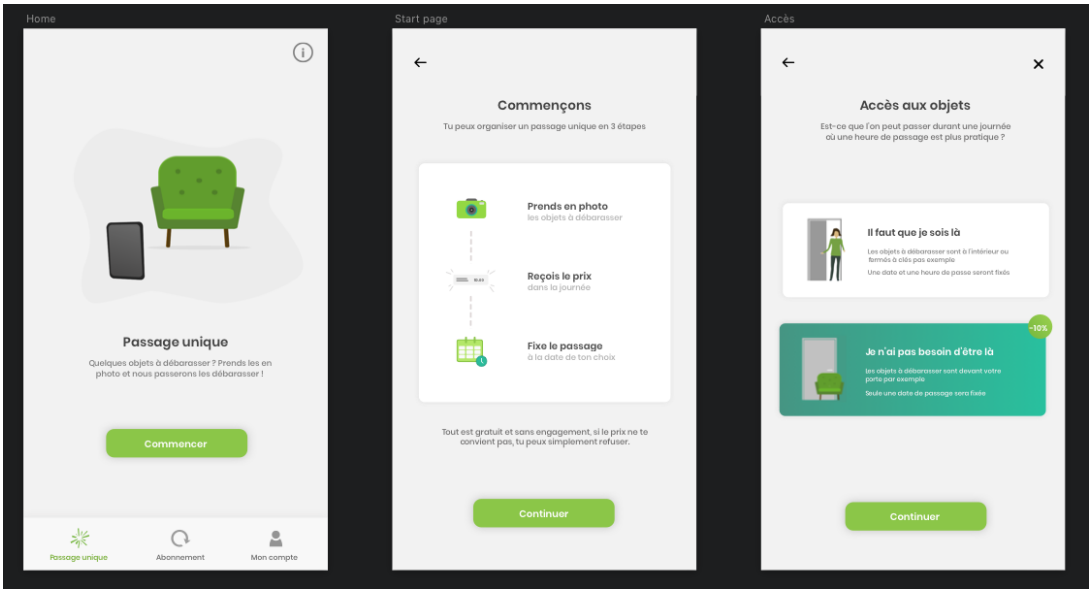

Fig. 4.3.: UI Design in Sketch

4.4. Implemented UI

Before detailing how the app was implemented in the next chapter, we will go through the final UI of the finished app and explain how its main parts function.

4.4.1. Lifts

The app is divided into two parts, the first one being lifts or one-time pickups. The idea is simple: customers can take a picture of anything they would like to dispose of, receive a quote for the elimination and book an appointment for it.

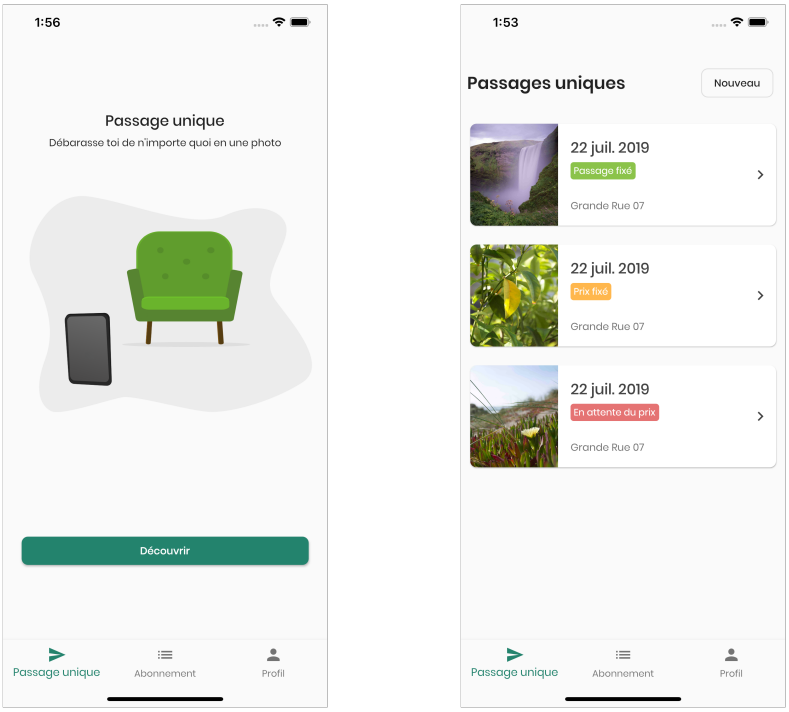

Fig. 4.4.: Lifts - Home page

Figure 4.4 shows the home page of the lift tab, on the left when the user is not authenticated or has not booked any lift yet, and on the right when there are existing lifts.

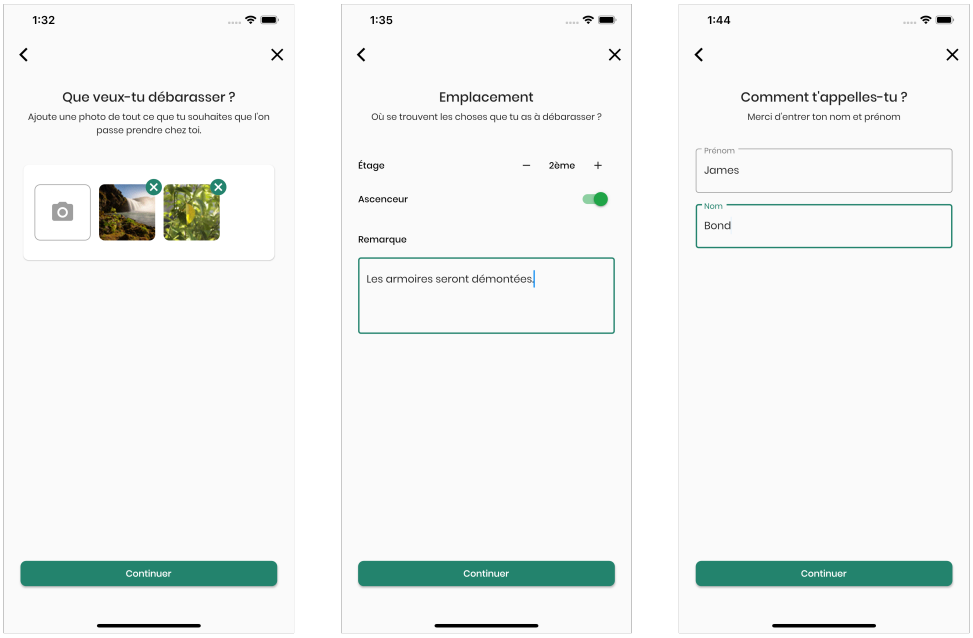

Fig. 4.5.: Lifts - Quote form 1

When the user clicks on *discover* or *new* a wizard form starts. On the first step, he is asked to provide pictures of what he wants to discard. The second step collects information about the location, such as the number of floors and whether there is an elevator or not. On the next step, personal information like name and address are asked. If they have already been inputted while creating another lift of subscription, those fields will be pre-filled.//

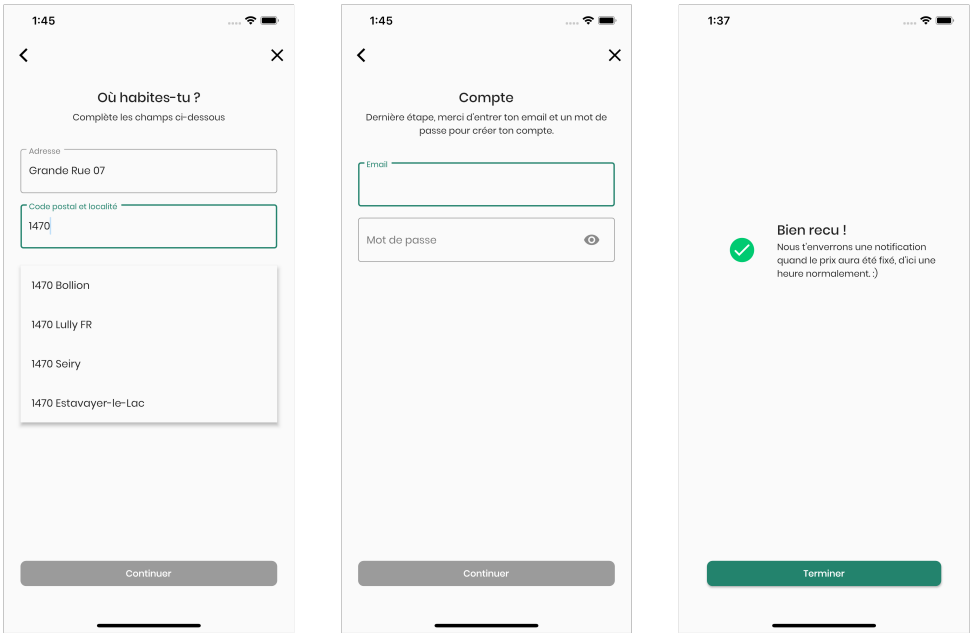

Fig. 4.6.: Lifts - Quote form 2

Postal codes are downloaded from the database and are searchable in the corresponding field. If a user is located in a postal code that is not covered, he will be redirected to a screen asking him if he wants to be contacted when we start serving his area. If the postal code is covered and the user is not authenticated, the form will ask for an email and password to create an account. If the user is authenticated, this step will be skipped.

When all steps are completed, a confirmation screen, as shown in the last screen of Figure 4.6 is displayed. At this point, all the data entered into the form is sent to the backend. Employees are then notified that a new lift has been created.

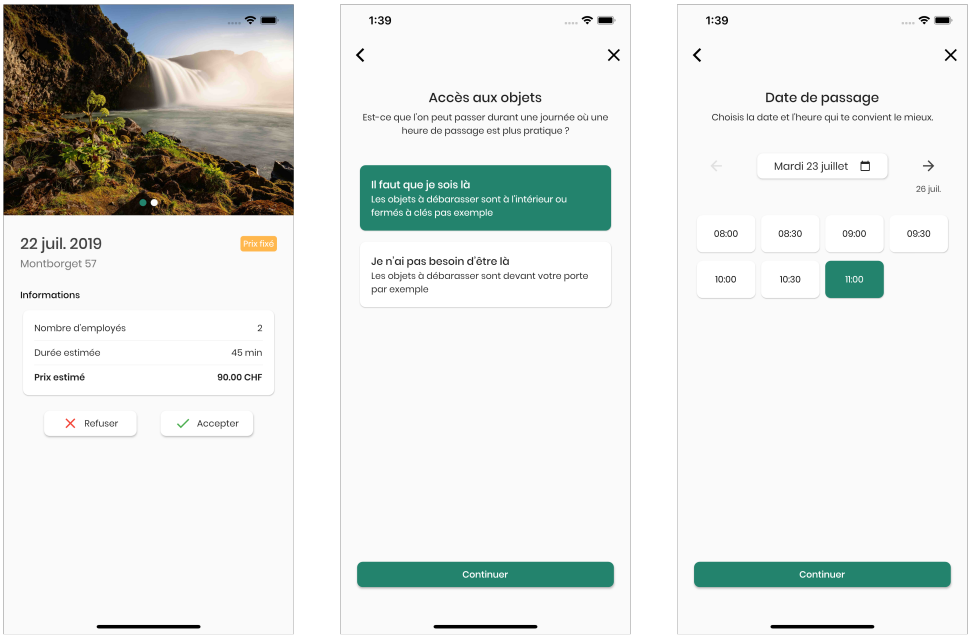

Fig. 4.7.: Lifts - Booking form 1

As soon as a price is set in the backend, a push notification is sent to the customer to inform him. When he clicks on the notification, the first screen of Figure 4.7 is displayed, and he can choose to accept or refuse the price that was set. If he accepts, the booking form starts. He is first asked whether he as to be present for the lift or not. This will determine if a date and time are scheduled, or just a date if he does not need to be present.

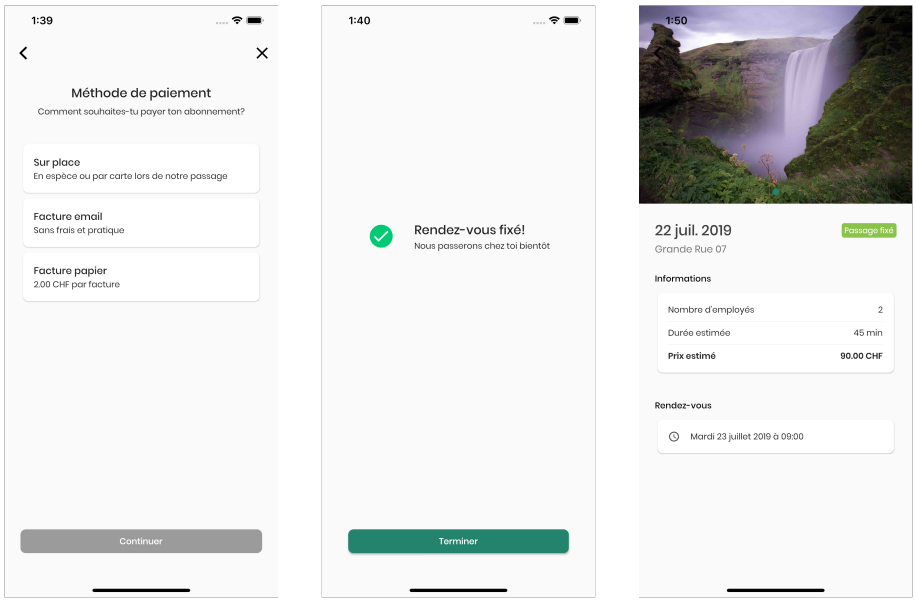

Fig. 4.8.: Lifts - Booking form 2

On the last step, the user can choose his payment method and then receives a confirmation message. Finally, the lift is set at the requested time.

4.4.2. Subscription

The second part of the app is dedicated to the customer's subscription. Here, users can manage their subscription and its pickups. If they do not have a subscription, they can create one through a wizard form similar to the one used to book a lift.

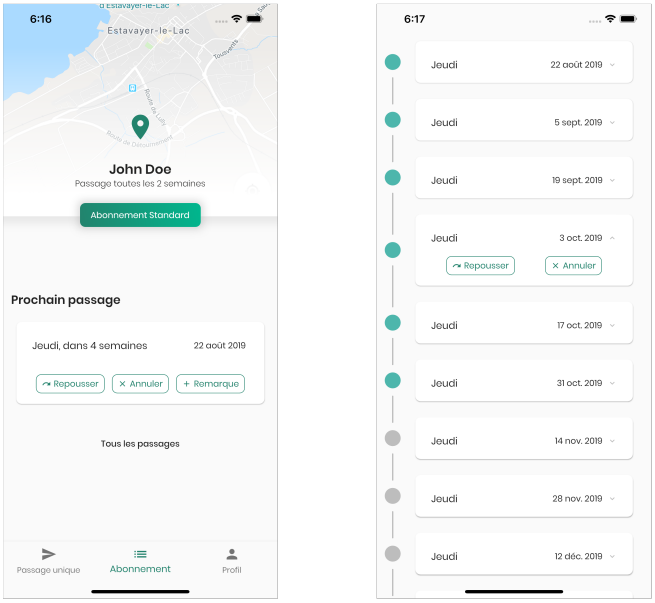

Fig. 4.9.: Subscription - Home screen

Figure 4.9 shows the home screen of a user who has a subscription, as well as the list of all pickups. For each pickup, the user can push it back, cancel it, or add a note. The blue dot on the left of the first six pickups indicates that they are part of the current subscription period and have been paid.

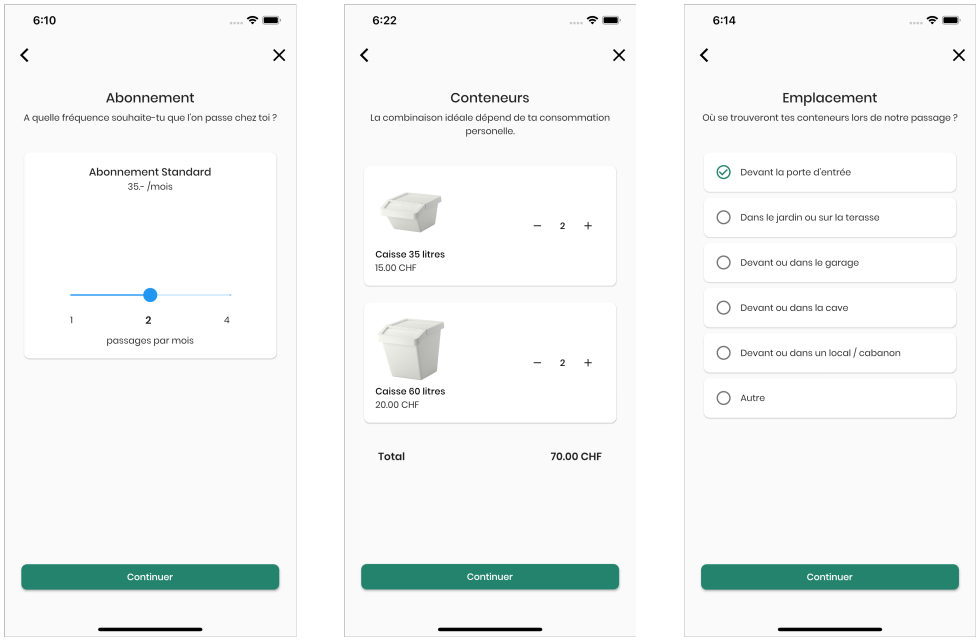

Fig. 4.10.: Subscription - Registration form 1

To create a subscription, the form guides the user through the registration process by first asking which frequency he wants. Some screens that appear next are not shown for the sake of brevity, such as name and address. The user can order sorting bins directly during sign up if he wants to. He is then asked where the sorting bins will be located.

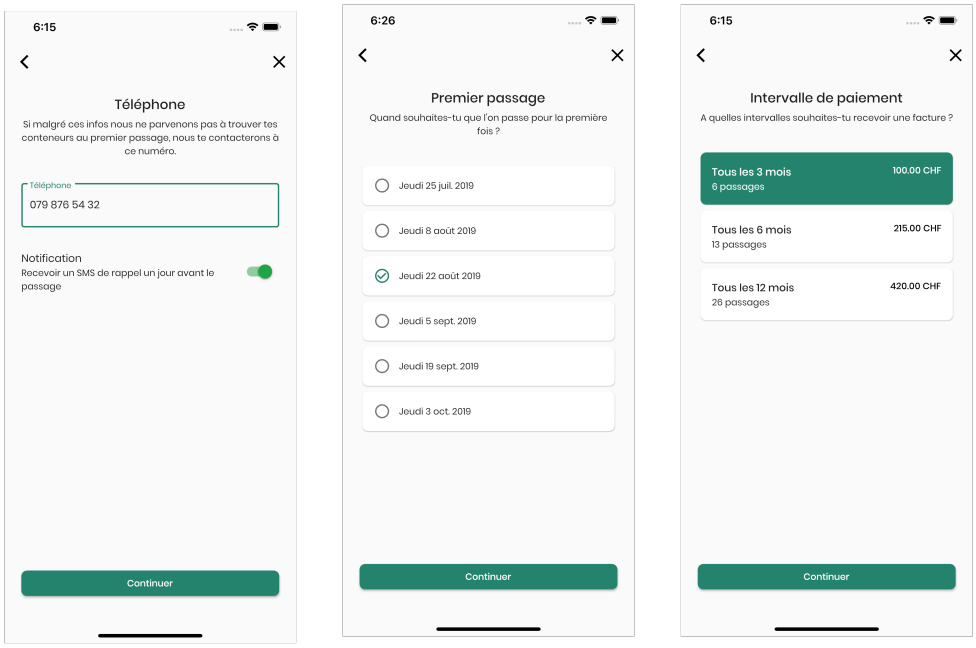

Fig. 4.11.: Subscription - Registration form 2

A phone number must be provided in case the sorting bins cannot be found, as well as to send optional SMS reminders one day before the pickup. The user can then choose the date of the first pickup from a list of possible start dates. At the end, the preferred payment method and the payment interval can be selected, after which the subscription is created. The resulting screen is the first displayed in Figure 4.9.

5 Development

5.1. Flutter

Flutter apps are written in Dart, a language developed by Google. It is an object-oriented statically typed language taking inspiration from many other languages, including Javascript and C [32]. Dart can be both AOT (Ahead of Time) and JIT (Just in Time) compiled, making fast development cycles possible with a feature called hot reload.

Unlike many other tools used to build apps, Flutter does not have a layout language like JSX or XML, but only Dart code. Flutter uses a declarative style of programming in opposition to most other UI frameworks such as Android SDK or iOS UIKit [31].

In Flutter, everything is called a widget. The idea is that each widget is a small element of the UI that can be composed with many others to build an app. As Flutter follows the declarative style, widgets describe what the UI should look like for a given state. When the state changes widgets are rebuilt to reflect the new state of the application. For example, a setting switched from *off* to *on* will trigger a rebuild of the interface.

```dart
import 'package:flutter/material.dart';

void main() {
  runApp(
    Center(
      child: Text('Hello, world!'),
    ),
  );
}
```

List. 5.1: Hello, world! in Flutter

Listing 5.1 shows a "Hello, world!" example in Flutter, composed of two widgets: Center and Text. Here the only state is the string "Hello, world!", and it does not change. In a real-world app, however, state changes a lot and managing it is one of the major challenges of building a mobile application.

5.2. State management

First, we need to clarify what is meant by state in this context. The Flutter documentation says that state is "whatever data you need in order to rebuild your UI at any moment in time" [33]. We can divide state into two types: local and global state.

1. **Local state**: This state is usually needed by a single widget, such as the progress of an animation. It is also called ephemeral state and can be assimilated to the RAM in a computer: used now and discarded later.
2. **Global state**: Several widgets generally share this state, and it is not ephemeral. Also called application state, it can be data related to the user or data that should be preserved through time.

5.2.1. Local state

Local state is relatively easy to manage through a widget included in Flutter: StatefulWidget[1]. Figure 5.1 shows an example of a screen using a StatefulWidget. This screen allows the user to choose when he wants us to pick up something. Several possible appointment times are displayed for July 19th, and the arrows can be used to navigate to later dates and their corresponding available times. Storing the currently displayed date is ideal for a StatefulWidget as this is purely ephemeral.

[1]StatefulWidget class: `https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html` - Accessed 31/07/2019

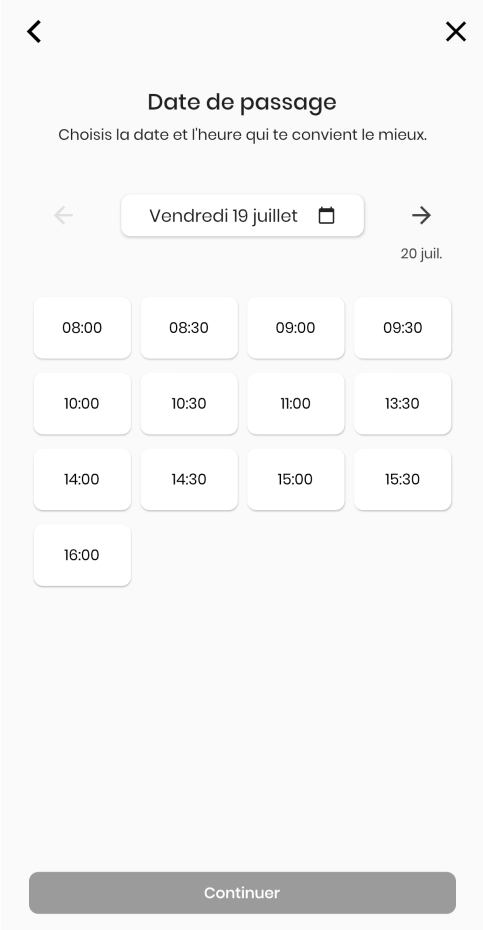

Fig. 5.1.: Slot picker screen

Listing 5.2 is a shortened version of the code of the slot picker. The full version is available on the GitHub repository. We can see that the SlotPicker widget extends StatefulWidget, and creates its state through `_SlotPickerState`. It takes a viewmodel as an argument, which we will see more in detail later as it gives access to the global state. In this case, the viewmodel makes available the list of dates and times available.

```
1  class SlotPicker extends StatefulWidget {
2    final _ViewModel viewModel;
3    SlotPicker(this.viewModel);
4
5    @override
6    _SlotPickerState createState() => _SlotPickerState();
7  }
8
9  class _SlotPickerState extends State<SlotPicker> {
10   List<DateTime> dates;
11   int currentDate = 0;
12
13   @override
14   Widget build(BuildContext context) {
15     return Column(
16       children: <Widget>[
17         Row(
18           children: <Widget>[
```

```
19            DateNavigationButton(
20                direction: Direction.previous,,
21                isActive: currentDate > 0,
22                onPressed: _previousDate),
23            RaisedButton(...), // Button between the two arrows
24            DateNavigationButton(
25                direction: Direction.next,
26                isActive: currentDate < dates.length - 1,
27                onPressed: _nextDate),
28          ],
29        ),
30        GridView(...) // Grid of available times
31      ],
32    );
33  }
34
35  void _previousDate() {
36    setState(() {
37        currentDate--;
38    });
39  }
40
41  void _nextDate() {
42     setState(() {
43        currentDate++;
44    });
45  }
46
47  @override
48  void initState() {
49    dates = toUniqueDates(widget.viewModel.liftSlots.toList());
50    super.initState();
51  }
52 }
```

List. 5.2: Slot picker code

The two pieces of local state here are a list of dates `dates`, and the index of the currently selected date `currentDate` initialized at 0 (line 10 and 11). The `dates` list is initialized each time the state object is created through `initState()` at line 48, which takes the complete list of available slots from the global state and returns a list of unique dates for our slot picker.

The `currentDate` is modified by the `_previousDate()` and `_nextDate()` functions by calling `setState()`, which notifies Flutter that the state changed, triggering a rebuild of the interface. Those functions are passed to `DateNativationButton`, a widget we created ourselves to display the arrow buttons, that calls the corresponding function when pressed.

5.2.2. Global state

Global state is not as straightforward, and there are many options available to manage it. There is no consensus on the best method to use, and the most appropriate option generally depends on the complexity and type of the application, as well as personal preference. Here are the main state management approaches used in Flutter:

- InheritedWidget and InheritedModel
- Provider and Scoped Model
- Redux
- BLoC / MobX

We will not go into details about each one of them, but some characteristics can be useful. InheritedWidget and InheritedModel are generally considered the most straightforward methods and are great for a small, not too complex app. Provider is relatively new and has recently been pushed by the Flutter team as a good default option when first starting out. The options are evolving rapidly, as Provider did not exist when we started this project. The remaining approaches: Redux, BLoC, and MobX, are generally used for larger and more complex apps. We quickly realized that an approach best suited for more complex apps would be a better option for us, leaving us with some leeway to grow as we hope our app will expand in functionality in the future. After some research and tests, we decided on Redux as it fits well with the type of app we have. It is also very popular in its original language Javascript, and there are therefore many resources about Redux available.

5.3. Redux

Flutter does its job very well: create user interfaces that run on multiple platforms. However, it does not do much more, to handle data and logic inside the app we need something else, which we have chosen to be Redux [34]. As a result, a large part of the codebase is based on the Redux architecture. We will first explain the basics of how it works so that the following sections are more comfortable to follow.

Redux was created in 2015 in Javascript, where it is widely used to manage state in web and mobile applications. As it is a well-proven architecture, it has been ported to the Dart language [35] and has a specific package for Flutter [36].

5.3.1. Redux cycle

Redux is based on a unidirectional data flow, as illustrated in Figure 5.2.

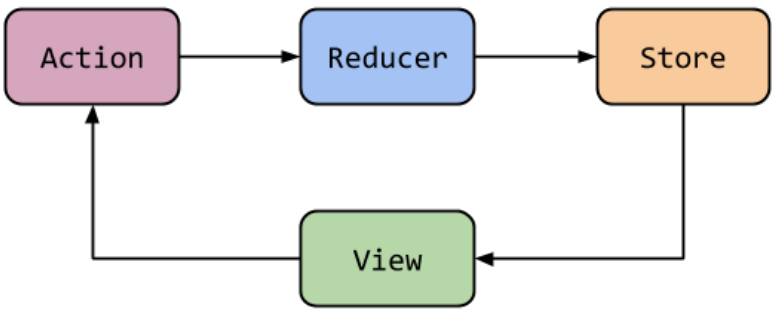

Fig. 5.2.: Redux cycle overview [37]

Redux follows three core principles that explain this cycle:

1. **Single source of truth** (Store): The global state of the app is stored in a single object called a Store.
2. **State is read-only** (Actions): The only way to modify the state is by emitting an action, which is an object describing what happened.
3. **Changes are made with pure functions** (Reducers): The state is updated by a Reducer: a function taking the previous state and an action as an argument, and returning a corresponding new state.

Let's take an example: a user switches bottom tabs in the app. The index of the active tab is in a part of the Store we named `NavState` as shown in Listing 5.3. Line 2 declares `selectedBottomNav`, the integer that stores the corresponding value. We store data as immutable values, but we will see more on that later.

```
1 abstract class NavState implements Built<NavState, NavStateBuilder> {
2   int get selectedBottomNav;
3
4   factory NavState() => _$NavState((NavStateBuilder b) => b
5     ..selectedBottomNav = 0
6   );
7 }
```

List. 5.3: Navigation State

When the user clicks on a new bottom tab, `ChangeBottomNavAction` (Listing 5.4) is dispatched, with the index of the tab that was clicked as payload. `ChangeBottomNavAction` is a simple Dart class, nothing more.

```
1 class ChangeBottomNavAction {
2   final int index;
3   ChangeBottomNavAction({this.index});
4 }
```

List. 5.4: Navigation Action

When `ChangeBottomNavAction` is dispatched, the Reducer function `_changeBottomNavReducer` (Listing 5.5) is called with the state `NavState` and the corresponding action as arguments. It return a new state with the `selectedBottomNav` value replaced by the value contained in the payload of the action: the tab that the user clicked.

```
1 NavState _changeBottomNavReducer(NavState state, ChangeBottomNavAction action) {
2   return state.rebuild((NavStateBuilder b) => b
3     ..selectedBottomNav = action.index,
4   );
5 }
```

List. 5.5: Navigation Reducer

At this point, the Store has been updated with the new state. The UI is rebuilt with the new tab active. This might seem like a lot of code to accomplish a straightforward action, and in this case, it is. One of the disadvantages of Redux is that it sometimes requires a high percentage of boilerplate code. On the other hand, this pattern starts to make sense as the application grows in complexity. It becomes easy to see the result of each action, and even to travel through time and see step by step what happened in the app.

5.3.2. Redux middleware

The cycle we have seen has one element missing: asynchronicity. When we need to fetch data from our server, for example, we cannot do it from Reducers as they should be pure functions. To handle this, we add a new step to the cycle called Middleware.

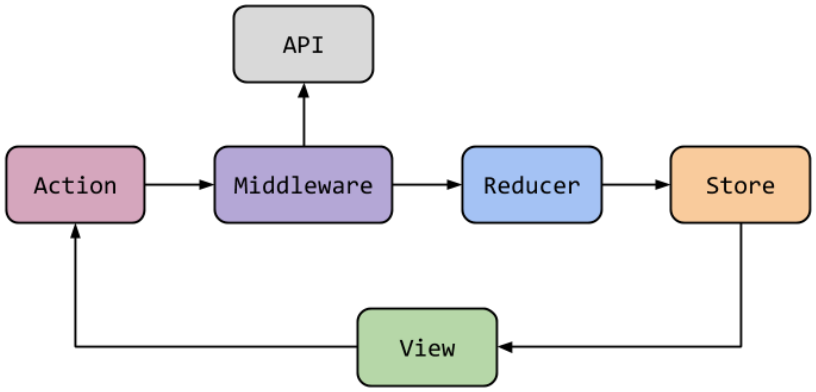

Fig. 5.3.: Redux cycle with Middleware [37]

Middlewares intercept actions before they get to Reducers. They can perform any operation, then dispatch the original action, another one or none at all. In our app, Middlewares are frequently used as we need to communicate regularly with our API. We can take the screen we saw earlier in Figure 5.1, where users can choose the date and time of a pickup, as an example. An action is dispatched a few screens before the widget is displayed, to fetch a list of timeslots available. As a result, the most up to date data is generated by the server and added to our Store. The action in question is `LoadLiftSlotsRequest` (Listing 5.6, line 1).

```
1  class LoadLiftSlotsRequest {}
2
3  class LoadLiftSlotsSuccess {
4    final List<DateTime> liftSlots;
5    LoadLiftSlotsSuccess({this.liftSlots});}
6
7  class LoadLiftSlotsFailure {
8    final String error;
9    LoadLiftSlotsFailure({this.error});
10 }
```

List. 5.6: Lift slots Actions

When the action is dispatched, `_loadLiftSlots()` (Listing 5.7) is called as it is set up to intercept `LoadLiftSlotsRequest` actions. We have a `repository` class containing `fetchLiftSlots()`, as well as a function for every endpoint we need to use on our API. It takes as argument the ID of the currently selected lift, which will be used in the URL of the request. In line 5, the JSON data is deserialized to a Dart list of datatimes. Then the Middlewareare dispatches a new action, `LoadLiftSlotsSuccess` (Listing 5.6, line 3) with the fetched list as argument. If an error occured at any point during those steps, `LoadLiftSlotsFailure` (Listing 5.6, line 7) is called instead with the corresponding error.

```
1  void _loadLiftSlots(Store<AppState> store, LiftBookFormStart action, NextDispatcher
       next) async {
2    next(action);
3    try {
4      final dynamic data = await repository.fetchLiftSlots(store.state.liftState.
           selectedId);
5      final List<DateTime> liftSlots = List<DateTime>.from(data.map<dynamic>((dynamic x)
           => DateTime.parse(x)));
6      store.dispatch(LoadLiftSlotsSuccess(liftSlots: liftSlots));
7    } catch (e) {
8      store.dispatch(LoadLiftSlotsFailure(error: e.toString()))
9    }
10 }
```

List. 5.7: Lift slots Middleware

Let's clarify the order of what happens. The Middleware (Listing 5.7) passes the original action to the Reducer with `next(action)` at line 2. The Reducer processes this action, and only then does the fetching of the data begin. Therefore in the Reducer, first `_loadLiftSlotsRequestReducer()` (Listing 5.8, line 1) gets the `LoadLiftSlotsRequest` action, and as a result returns a new state with the `isLoading` flag set to true. Then if the data fetching was successful, `_loadLiftSlotsSuccessReducer()` is called with the list of datetimes as payload of the action. The `isLoading` flag is set to False, and the list of time slots are added to the new state. If the data fetching isn't successful, then `_loadLiftSlotsFailureReducer()` is called, `isLoading` is set to False and the error message is saved to the Store.

```
1  LiftBookFormState _loadLiftSlotsRequestReducer(LiftBookFormState state,
       LoadLiftSlotsRequest action) {
2    return state.rebuild((b) => b..isLoading = true);
3  }
4
5  LiftBookFormState _loadLiftSlotsSuccessReducer(LiftBookFormState state,
       LoadLiftSlotsSuccess action) {
6    return state.rebuild((b) => b
7      ..isLoading = false
8      ..liftBookForm.replace(state.liftBookForm.rebuild((b) => b
9        ..liftSlots.replace(action.liftSlots)
10     )));
11 }
12
13 LiftBookFormState _loadLiftSlotsFailureReducer(LiftBookFormState state,
       LoadLiftSlotsFailure action) {
14   return state.rebuild((b) => b
15     ..isLoading = false
16     ..error = action.error);
17 }
```

List. 5.8: Lift slots Reducers

5.4. Implementation challenges

Now that we have covered the two main building blocks of our app, Flutter and Redux, we will go through the main challenges we encountered during the development of the app.

5.4.1. Data modeling

As we have around twenty different data models such as user, customer, and subscription, we need an efficient way to serialize and deserialize data from the API. We could write the functions to do it by hand, but this can take a long time, and there are more efficient ways of doing this.

One of them is a package called built_value [38]. The primary purpose of this package is to provide immutable value types. The term value type is used to describe types where equality is based on value, not on a reference. A 6 is always equal to another 6 and cannot be changed. They are immutable by nature.

That property goes hand in hand with Redux, which for various reasons usually requires immutable data types [39]. What is interesting about the built_value package is that it comes with serialization and deserialization built-in. If we take our user data model as an example, it is declared in around 10 lines of code, and 140 corresponding lines of code are generated by built_value. The generated code includes serialization functions, and many other repetitive functions we would otherwise have to write ourselves.

5.4.2. Wizard forms

We had a few requirements in mind for the wizard forms before we started building them. As they are quite long and ask for numerous information, the user should be able to leave the form and come back to it without losing data he entered. Achieving data persistence is not a trivial task. In our case, it means each time a letter is tapped or an option is chosen, data should be saved to the Redux Store. We did this by creating a data model for each form that includes every answer as fields. At every input event, an action is dispatched with the new value as payload. A corresponding Reducer updates the Store with the new information. Form fields retrieve data saved in the Store when they are built so that the data inputted is still here if the user leaves the form.

Another requirement is that the user should be able to navigate back and forth between screens freely. As some screens are conditional on previous input, the *back* and *next* button will not always link to the same page. There are several ways we could have built this, but we choose to use Flutter built-in Navigator [40] along with Redux. An integer called step is saved in the Store. When the next button is tapped, an action is dispatched, the Reducer increments step by 1, and a Middleware navigates to the screen corresponding to the current step. The Reducer also checks the values currently in the Store, and can conditionally increment by a different number if a screen has to be skipped based on the user's previous choices. For example, if a user said he does not want sorting bins on screen 7, the Reducer is going to increment the step to 9 if screen 8 is the sorting bins screen. Forward navigation works by pushing a new route onto a navigation stack. As a result, the back button simply pops the current route to get to the previous screen.

The last problem is that if the user closes the form, every route pushed to the navigation stack is popped until only the home tab is left. If the user starts the form again, and we just navigate to the screen corresponding to the current step, back navigation would be lost. This is because the routes of the previous screens would not be in the navigation stack. To solve this, a Middleware checks the current step when the user starts or returns to a form. It then pushes every route from the first step of the form to the current step onto the navigation stack. In doing so, it also skips screens that should not be in the stack based on previous choices of the user. This results in the same navigation stack than if the user never left the form, where he can freely navigate between screens.

5.4.3. Push notifications

Having a way to contact our users after they have requested a quote for a lift is critical. If we do not have a reliable way to contact users, we may lose potential customers. That is the purpose of notifications, which can be either local or push. Local notifications are programmed by the app to be sent at a specific time. Push notifications are sent from a server at any time, and they are the type we need as prices are set at an undetermined time.

Android and iOS both have different ways of managing push notifications. Luckily there is a service offered by Firebase called Firebase Cloud Messaging (FCM) that simplifies the process. Some platform-specific setup has to be done on the Flutter app for FCM to be used, but it is relatively straightforward. Once this is done, the process is the following:

- On the app, retrieve a device registration token. This is a unique identifier linked to the current install of the app.
- When the user submits a lift, save the current token on the backend and link it to the user that submitted the lift.
- After a price is set for the lift, send a push notification from the backend to the corresponding token.

For both the Flutter app and the python backend, there is an FCM package [42] or library [43] available.

5.4.4. Image storage

Ideally, we would have liked to store the images from the lift request on our existing servers. After some research, we found out that storing images directly in a relational database such as PostgreSQL is not optimal. Instead, it is generally considered a better approach to store images in dedicated hosts such as Amazon S3 or Firebase.

There are two main methods to get the image to a dedicated host [41]. The first one, called pass-through, consists of creating an endpoint on the backend REST API, which in turn uploads the image to the chosen host. However, this method has several drawbacks, such as latency issues and timeout problems on the backend.

Instead, we chose the second option: direct uploads. Here the image is sent directly to the host, which returns a URL to the image. We then store the URL in the database. Since there is excellent support from Firebase in Flutter, we chose this provider as host for our images. When the user selects an image, it is immediately uploaded by the Firebase

plugin. When the upload is complete, a URL is returned, and an action is dispatched with the URL as payload. A Middleware then creates a Lift Image object in the database with the URL of the image and the backend assigns it an ID. The app saves both the URL used to fetch images from Firebase and the ID to manage them on our backend.

5.4.5. Lift appointment timeslots

After customers have accepted the price of a lift, they can choose the date and time of the appointment. To generate this list, we need to take different scheduling information into account. First, appointment slots should be given during *Oust!* opening hours. They should also be proposed when the recycling center closest to them is open. Moreover, at least one vehicle must be available during the expected duration of the lift.

Scheduling information can be very tricky to store and manage. Opening hours of recycling centers, for example, are recurring events that can be modified if they happen to be on a public holiday. This gets quickly complicated to implement if we decide to store that data on our database. What's more, we then need to create a user interface to display existing appointments and modify them as some are added after a phone call or email.

We decided instead to store all scheduling information on Google Calendar. It is already used in the company because the mobile and web app to consult appointments are excellent. There is an API through which we can easily create and manage calendars from our backend through a Google service account. As a result, we have a separate calendar for our opening hours and opening hours of recycling centers. The backend also automatically creates two calendars per vehicle, one for lift appointments and one for scheduled rounds.

To generate the list of available timeslots, we use simple operations that can be assimilated to set theory. We fetch each event of a given calendar and transform them in timeslots, which are simple instances with a start and end datetime. We have defined two operations: intersection and difference, as shown in Listing 5.9.

```
class TimeSlot:
    def __init__(self, start, end):
        self.start = start
        self.end = end

    # Returns true if a and b overlap
    def overlap(a, b):
        return a.start < b.end and b.start < a.end

    # Returns true if a is contained in b
    def contains(a, b):
        return a.start >= b.start and a.end <= b.end

    # Returns all timeslots_a contained in any timeslot_b
    def intersection(timeslots_a, timeslots_b):
        result = []
        for timeslot_a in timeslots_a:
            for timeslot_b in timeslots_b:
                if TimeSlot.contains(timeslot_a, timeslot_b):
                    result.append(timeslot_a)
                    break
        return result
```

```
23
24    # Returns all timeslots_a not overlapping with any timeslot_b
25    def difference(timeslots_a, timeslots_b):
26        result = timeslots_a.copy()
27        for timeslot_a in timeslots_a:
28            for timeslot_b in timeslots_b:
29                if TimeSlot.overlap(timeslot_a, timeslot_b):
30                    result.remove(timeslot_a)
31                    break
32        return result
```

List. 5.9: Scheduling set operations

When we set the price of a lift, we also decide whether it should be completed during a round if it is small, or during a day without rounds otherwise. Possible timeslots are generated from the estimated duration and an appointment interval of 30 minutes, for example. Then we take the difference or intersection of the sets of timeslots as needed to get to the list of available appointment times, as shown in Table 5.1.

Calendars

- $P = \{$ Possible timeslots $\}$
- $O = \{$ *Oust!* Opening hours $\}$
- $C = \{$ Collection Center Opening hours $\}$
- $R = \{$ Rounds scheduled $\}$
- $L = \{$ Lifts scheduled $\}$

Lift outside of rounds	$P \cap O \cap C \setminus R \setminus L$
Lift during a round	$P \cap O \cap C \cap R \setminus L$

Tab. 5.1.: Operations for availability timeslots

5.4.6. Subscription start dates

Up until now, as customer's subscriptions were manually created during an appointment, we would choose the starting date by grouping customers in the same area. If a subscription is created through the app, this grouping must be done by the backend with an algorithm.

This is how it works:

1. Get the GPS coordinates of the new subscription-based on its address using Google Geocoding [44].
2. For every future round of this postcode, get a driving duration matrix from the new subscription to every customer in the round using OSRM. Save the lowest value, which is the closest customer, as an estimate of the time added to the round.
3. Iterate over every future round of this postcode. For each, sum up the additional times from step 2 over the next 4 pickups. For example, if the current date iterated

is July 1 and the subscription is for a pickup every 2 weeks, get the sum of the additional time for July 1, July 15, July 29, and August 12. This gives us an estimate of the impact on rounds duration if the new subscription were to start on July 1.

4. Group the result of step 3 for every future round of this postcode using mean shift clustering.
5. Order the clusters of step 4 from the lowest impact on rounds to the highest and add the clusters of dates to the result following this order until there is at least one start date proposed per month.

We had to take a few points into account when developing this algorithm. First, there are different frequencies of pickups: a subscription can include a pickup every 1, 2 or 4 weeks. As a result, calculating the additional time the new subscription would add to a single round does not reflect the whole picture. We have to check the impact a new subscription has on a given round and the rounds coming after it.

In this algorithm, we use the driving duration to the closest customer in the round as an approximation of the additional time required to complete the pickup. The ideal method would be to optimize the route of the round with and without the new customer to get a better estimate. We have an optimization model specifically designed for our rounds and customers, but it takes at least several seconds to complete. Since the request for possible start dates is made by the mobile app a few seconds before the list of dates is displayed, this is too long. Getting a driving duration matrix from OSRM, on the other hand, takes less than 100 milliseconds.

Finally, the calculation of the impact of different start dates on rounds is grouped in clusters for a specific reason. If we were to return the start dates lower than the median or the mean, some start date just over the threshold would be similar to those lower and should be proposed, but would not be. By clustering the results and returning clusters of values, we ensure that the largest possible number of optimal start dates are offered to the user. We use a mean shift method for the clustering itself, from the sklearn package [45]. Mean shift does not need to be given the number of clusters to be found, which is ideal in our case as there can be an arbitrary number of them.

6 Conclusion

When we started *Oust!*, I would never have thought such an incredible learning process would follow. Having a practical experience alongside my studies has been a fantastic way to put into practice theoretical knowledge from courses. That is also the case because the other co-founders and I have been fortunate enough to find professors that welcomed projects dear to our hearts with an open mind. In the case of this report in particular, I am very thankful to the professors and assistants that supported it even though my bachelor's degree is in Economics.

I learned about software development by necessity to allow our company to function. In the process, I discovered an intense interest in the subject, and this knowledge is invaluable for guiding the future of my professional life. Developing this app has been a lot of work, and there is still a lot more to be done. Many small but critical pieces have to be put together carefully. At the time of writing, the app is still a few weeks away from being launched as we want to test it with early users and iron out a few details. If the app has some success after the launch, we are full of ideas to implement and ways to improve what was built up until now.

In the end, I am very happy with the result of this project. I think that tools such as Flutter open the doors of building mobile applications to a much larger number of people than ever before. The ability to build an app in one codebase and one language is key for small companies or individuals that want to test out an idea. I hope it brings lots of innovation to a world that, in some domains, urgently needs it.

A

Common Acronyms

API	Application Programming Interface
AWS	Amazon Web Services
AOT	Ahead Of Time
CRUD	Create, Read, Update and Delete
DRF	Django Rest Framework
FCM	Firebase Cloud Messaging
HTTP	Hypertext Transfer Protocol
JIT	Just In Time
JSON	JavaScript Object Notation
JSX	JavaScript XML
MRF	Material Recovery Facility
ORM	Object Relational Mapping
OSRM	Open Source Routing Machine
REST	Representational State Transfer
SDK	Software Development Kit
SQL	Structured Query Language
UI	User Interface
UX	User Experience
URI	Unified Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language

B

Repository of the Project

The mobile app will be available on Google Play and on the Apple Store.

The repository of the app is available here: `https://github.com/SylvainLosey/oust`. A majority of the code is located in the `lib` folder, in which can be found:

- Data: models of the different data structure as well as the repository.
- Redux: the business logic of the app is grouped in this folder
- UI: the files related to the user interface of the app
- Utils: various utility files

The code for the backend is not public but can be provided upon request.

References

[1] X. Shi, A. E. Thanos, and N. Celik, "Multi-objective agent-based modeling of single-stream recycling programs," *Resources, Conservation and Recycling*, vol. 92, pp. 190–205, nov 2014. 3

[2] C. Lakhan, "A Comparison of Single and Multi-Stream Recycling Systems in Ontario, Canada," *Resources*, vol. 4, no. 2, pp. 384–397, jun 2015. 3

[3] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, 2000. 11

[4] D. A. Norman, *The Design of Everyday Things*. New York, NY, USA: Basic Books, Inc., 2002. 18

[5] R. Molich and J. Nielsen, "Improving a human-computer dialogue," *Communications of the ACM*, vol. 33, no. 3, pp. 338–348, mar 1990. 18

Referenced Web Resources

[6] D. Carrington, "School climate strikes: 1.4 million people took part, say campaigners," The Guardian, Mar. 2019, Accessed 31/07/2019. [Online]. Available: https://www.theguardian.com/environment/2019/mar/19/school-climate-strikes-more-than-1-million-took-part-say-campaigners-greta-thunberg 1

[7] C. Katz, "How China's Ban on Importing Waste Has Stalled Global Recycling," Yale School of Forestry and Environmental Studies, Mar. 2019, Accessed 31/07/2019. [Online]. Available: https://e360.yale.edu/features/piling-up-how-chinas-ban-on-importing-waste-has-stalled-global-recycling 1

[8] "Coûts et prestations de la gestion communale des déchets," Infrastructures communales, Union des villes suisses et de l'Association des Communes Suisses, 2009, Accessed 31/07/2019. [Online]. Available: https://kommunale-infrastruktur.ch/cmsfiles/rapport_annexe.pdf 3

[9] "Rapport sur la gestion des déchets 2008," Office fédéral de l'environnement OFEV, 2008, Accessed 31/07/2019. [Online]. Available: https://www.bafu.admin.ch/bafu/fr/home/themes/dechets/publications-etudes/publications/rapport-gestion-dechets-2008.html 4

[10] T. Guardian, "Americans' plastic recycling is dumped in landfills, investigation shows," Accessed 31/07/2019. [Online]. Available: https://www.theguardian.com/us-news/2019/jun/21/us-plastic-recycling-landfills 4

[11] "The Web framework for perfectionists with deadlines | Django," Accessed 31/07/2019. [Online]. Available: https://www.djangoproject.com/ 9

[12] "Celery: Distributed Task Queue," Accessed 31/07/2019. [Online]. Available: http://www.celeryproject.org/ 10

[13] "Django Celery Beat," Accessed 31/07/2019. [Online]. Available: https://github.com/celery/django-celery-beat

[14] "Bexio: Business Software for Small Businesses and Startups," Accessed 31/07/2019. [Online]. Available: https://www.bexio.com/en-CH/ 10

[15] "Bexio API Documentation," Accessed 31/07/2019. [Online]. Available: https://docs.bexio.com/

[16] "Firebase," Accessed 31/07/2019. [Online]. Available: https://firebase.google.com/ 10

[17] "Google Calendar API," Accessed 31/07/2019. [Online]. Available: https://developers.google.com/calendar/ 10

[18] "Project OSRM," Accessed 31/07/2019. [Online]. Available: http://project-osrm.org/ 11

[19] "Amazon EC2," Accessed 31/07/2019. [Online]. Available: https://aws.amazon.com/ec2/ 11

[20] "Heroku," Accessed 31/07/2019. [Online]. Available: https://www.heroku.com/ 11

[21] M. Fowler, "Richardson Maturity Model - steps toward the glory of REST," Mar. 2010, Accessed 31/07/2019. [Online]. Available: https://martinfowler.com/articles/richardsonMaturityModel.html 12

[22] "Swagger UI," Accessed 31/07/2019. [Online]. Available: https://swagger.io/tools/swagger-ui/ 14

[23] "drf-yasg - Yet Another Swagger Generator," Accessed 31/07/2019. [Online]. Available: https://github.com/axnsan12/drf-yasg 14

[24] G. Peal, "Sunsetting React Native," Jun. 2018, Accessed 31/07/2019. [Online]. Available: https://medium.com/airbnb-engineering/sunsetting-react-native-1868ba28e30a 17

[25] "Flutter - Beautiful native apps in record time," Accessed 31/07/2019. [Online]. Available: https://flutter.dev/ 17

[26] StackOverflow, "Developer Survey Results," 2019, Accessed 31/07/2019. [Online]. Available: https://insights.stackoverflow.com/survey/2019 18

[27] "Get ahead in UI/UX Design - Video Courses - learnux.io," Accessed 31/07/2019. [Online]. Available: https://learnux.io/ 18

[28] J. Nielsen, "10 Usability Heuristics for User Interface Design," Apr. 1994, Accessed 31/07/2019. [Online]. Available: https://www.nngroup.com/articles/ten-usability-heuristics/ 18

[29] "Sketch - The digital design toolkit," Accessed 31/07/2019. [Online]. Available: https://www.sketch.com/ 20

[30] "Prototype of the App in Sketch," Accessed 31/07/2019. [Online]. Available: https://sketch.cloud/s/5Yb9E/a/ZLm0Dv/play 21

[31] "Introduction to declarative UI," Accessed 31/07/2019. [Online]. Available: https://flutter.dev/docs/get-started/flutter-for/declarative 28

[32] "Dart programming language," Accessed 31/07/2019. [Online]. Available: https://dart.dev/ 28

[33] "Ephemeral and app state," Accessed 31/07/2019. [Online]. Available: https://flutter.dev/docs/development/data-and-backend/state-mgmt/ephemeral-vs-app 29

[34] "Redux · A Predictable State Container," Accessed 31/07/2019. [Online]. Available: https://redux.js.org/ 32

[35] "Redux for dart," Accessed 31/07/2019. [Online]. Available: https://pub.dev/packages/redux 32

[36] "Flutter Redux," Accessed 31/07/2019. [Online]. Available: https://pub.dev/packages/flutter_redux 32

[37] X. Rigau, "Introduction to Redux in Flutter," Novoda, Apr. 2018, Accessed 31/07/2019. [Online]. Available: https://blog.novoda.com/introduction-to-redux-in-flutter/ v, 32, 34

[38] "built_value | Dart Package," Accessed 31/07/2019. [Online]. Available: https://pub.dev/packages/built_value 36

[39] "Immutable Data · Redux," Accessed 31/07/2019. [Online]. Available: https://redux.js.org/faq/immutable-data 36

[40] "Navigator class," Accessed 31/07/2019. [Online]. Available: https://api.flutter.dev/flutter/widgets/Navigator-class.html 36

[41] "Using AWS S3 to Store Static Assets and File Uploads," Accessed 31/07/2019. [Online]. Available: https://devcenter.heroku.com/articles/s3 37

[42] "Firebase Cloud Messaging for Flutter," Accessed 31/07/2019. [Online]. Available: https://pub.dev/packages/firebase_messaging 37

[43] "pyfcm," Accessed 31/07/2019. [Online]. Available: https://pypi.org/project/pyfcm/ 37

[44] "Google Geocoding API," Accessed 31/07/2019. [Online]. Available: https://developers.google.com/maps/documentation/geocoding/intro 39

[45] "Sklearn Mean Shift," Accessed 31/07/2019. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.MeanShift.html 40

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Faculté des sciences économiques et sociales
Wirtschafts- und sozialwissenschaftliche Fakultät
Boulevard de Pérolles 90
CH-1700 Fribourg

Fribourg, _____

D E C L A R A T I O N

I hereby declare that I wrote this thesis on my own and followed the principles of scientific integrity.

I acknowledge that otherwise the department has, according to a decision of the Faculty Council of November 11th, 2004, the right to withdraw the title that I was conferred based on this thesis.

I confirm that this work or parts thereof have not been submitted in this form elsewhere for an examination, according to a decision of the Faculty Council of November 18th, 2013.

……, the ..... 20.....

...……
(Signature)