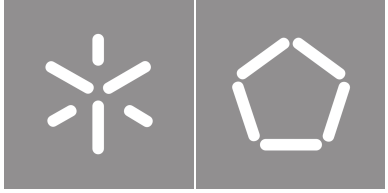
Universidade do Minho
School of Engineering

Filipe José da Silva Freitas

Wintouch Cloud - Delivery Module

October, 2022



Universidade do Minho
School of Engineering

Filipe José da Silva Freitas

Wintouch Cloud - Delivery Module

Master Thesis

Master in Informatics Engineering

Work developed under the supervision of:
Pedro Manuel Rangel Santos Henriques
Carlos Ribeiro

October, 2022

This document was created with the LuaLaTeX processor and the NOVAthesis template (v6.10.1) [4].

Acknowledgements

I would like to thank my advisor and professor, Dr. Pedro Rangel Henriques, for the valuable insights provided during the development of this thesis, as well as for the support and encouragement provided throughout the process, and without which this work would not have been possible.

I would also like to thank Wintouch for the opportunity granted to me to develop this work, for the support provided throughout the process, and for the valuable insights, knownledge, and experience shared with me by my colleagues.

A very special thanks to my friend Ana Macedo for her support and encouragement, and for letting me use her chat as a venting space and only responding half of the time.

I would also like to thank Pedro Pinheiro and Pedro Festa, for their encouragement and support throughout the process, without which this work would not have been possible.

Last but not least, I would like to thank my friends that were not mentioned, as well as my family for their support and encouragement throughout the process, and for listening to me and providing me with support during the development of this thesis.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

"Education is the most powerful weapon which you can use to change the world." (Nelson Mandela)

Abstract

Wintouch Cloud - Delivery Module

This document constitutes the final report of a Master's Thesis focused on developing a working software product for the company Wintouch, with the purpose of managing the *delivery* of prepared/cooked meals at restaurants.

The software product developed and here discussed (the working process and the final product) manages the entire process of deliveries in restaurants, allowing clients to place orders online, via a phone call or in person, and keeping track of the subsequent tasks until the order is delivered to the client. This management includes tasks such as deciding when to start preparing the order, sending the couriers to clients' homes, while managing their routes and maximizing the number of orders they take to a certain area. The software package and application developed and under discussion also ensures a proper interaction with Wintouch's products, allowing restaurants to save information about clients, so as to increase the efficiency of future contacts with clients.

Keywords: Home Food Delivery, Take-away, Restaurant, Web Development, Angular, .NET

Resumo

Wintouch Cloud - Módulo de Delivery

Este documento constitui o relatório de uma Tese de Mestrado focada no desenvolvimento de um produto de software para a empresa Wintouch, com o propósito de gerir as *entregas* de refeições preparadas/cozinhadas em restaurantes.

O produto deverá gerir todo o processo de preparar entregas em restaurantes, deixando clientes fazer pedidos online, por telefone ou presencialmente, e após o pedido ser realizado, gerir todas as tarefas subsequentes até que o pedido seja entregue ao cliente, tal como gerir quando começar a preparar a entrega do pedido, enviar estafetas para as casas dos clientes, gerir as suas rotas e maximizar o número de pedidos entregues numa determinada área. Deverá também interagir com os restantes produtos da Wintouch, permitindo que os restaurantes guardem informações sobre os seus clientes, de modo a maximizar a eficiencia dos contactos futuros com os mesmos.

Palavras-chave: Entrega ao Domicílio, Comida para levar, Restauração, Desenvolvimento Web, Angular, .NET

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The On-Demand Food Delivery market, i.e., "*the purchase and delivery of freshly prepared meals from restaurants to the customers' home enabled by the use of online platforms*" [6], has been steadily growing over the last few years in every region of the world, including Portugal, where user penetration is expected to double from 2018 until 2022 [5]. Needless to say, this extreme demand rise has been driven by modern smartphone apps [6], such as UberEats, Glovo, NoMENU, SendEAT, Takeaway.com, and *Comer em Casa* [5].

These modern apps must meet certain requirements to be successful. According to [2], the two most important success factors are waiting time and food quality, with food quality increasing in importance from 22.38% to 46.8% in 2016, according to [2]. Other factors consumers considered important were the design of the apps, the convenience of the process, and order accuracy [5]. These apps must also balance these factors (and others, such as food preparation time) with the cost of delivery [2].

Wintouch, the company that proposed this thesis, launched in the year 2000, released their first two products, wSIR and wPOS, now known as *WINTOUCH Restauranção* and *WINTOUCH Retalho*, with the first one being directly related to the restaurant area. Specifically, it was a software product for the management of restaurant's orders, keeping track of orders made, issuing invoices, integrating with the kitchen (showing, in a monitor, which orders need to be produced and when), among other things[1]. This makes Wintouch a pioneer in this industry in Portugal.

Their first line of products is still in use today. These products were based on Desktop applications built to be run locally, on the customer's premises. They were also built over the last two decades. That's why, in 2016, they launched a new line of cloud, web-based products. These products are built on top

[1] https://cms.wintouch.pt/index.php/company/whoweare

of modern web technologies, and because of this, allow for easier maintenance, and given that they are more modern, allow for features to be more easily implemented. The software product that this thesis will focus on will be built on top of this platform, as a separate module, meaning it will integrate with Wintouch Cloud's existing restaurant management solutions.

Unlike other established delivery apps, the module this thesis will focus on developing will take a different approach, specifically in terms of delivery drivers. Because Wintouch's Cloud solutions are already present at major restaurants in the country, this product will serve the purpose of connecting customers with the restaurants Wintouch already serves. However, the deliveries themselves will be made by the restaurant's own staff (either by hiring new drivers, or using already existing drivers). What this means, in practice, is that a customer will order their food through some platform (either online, via telephone, or another means), and, after the order is placed, the order is sent to the restaurant's internal systems (which, because they are already Wintouch Cloud clients, will also be made by Wintouch), where the food will be prepared, packaged, and delivered by the restaurant's own staff. These processes of managing the preparation, packaging and delivery of the order will also be implemented into the current Wintouch Cloud solution, so as to properly integrate with the new delivery module.

1.2 Objectives

This Master's thesis will be split into two parts: a smaller, more theoretical, initial part, followed by a large development stage.

There will also be two classes of objectives: the first class of objectives are this thesis' main objectives. These are all of the objectives that are expected to be fully completed by the end of this work. However, the second class of objectives, or secondary objectives, includes some objectives that, given the time and resource constraints of this thesis, might not be feasibly completed during this project. However, the company could pick them up and implement them at a later date.

As stated at the beginning, the main objective of this Master's Project is the development of an application to allow Wintouch's clients to have their food requests delivered to their homes efficiently, at the time they have requested.

To properly accomplish that objective, there are sub-objectives and requirements to complete, as follows:

- Allow the restaurant to insert a customer's order directly into the system, in case the order is placed in-person or via a telephone call;
- The back-office should keep information about customers, such as delivery addresses, customer order history, among others;
- The back-office will manage when orders will begin cooking, according to the requested delivery time, the time it takes for the order to be made, the time it takes to deliver, etc;

- The back-office should manage the delivery drivers that will be making the deliveries, maximizing the number of orders delivered each time the driver goes out to an area, among other optimizations.

According to the explanation above, there are also secondary objectives or requirements for this Master's Project's work. They are:

- Use automatic delivery zone mapping;

Other requirements may be added or removed throughout development. They might also be adjusted, after taking into account further literature review, client interviews, and other sources.

Lastly, as a professional software development company, Wintouch expects this software product to be built according to the highest industry standards, with proper documentation and support, such that training can then be provided to Wintouch's commercial partners.

The next sub-section explains the secondary objective listed above in more detail.

1.2.1 Delivery Zone mapping

Delivery Zones are geographical areas where restaurants may or may not deliver food to at certain times. Their geographical limits are, in Wintouch's Desktop products, defined manually, by inputting every address in a certain region into the list of addresses that belong to that zone. However, there are processes to automate this, specifically, by using official lists of addresses that belong to certain parishes, and allowing users to specify which parishes should be used to fill in the zone's addresses.

In this Cloud product, we intend on using a different approach. Google provides their own Maps JavaScript API, which you can use to, among other things, draw areas on maps. We intend on utilizing this feature to allow restaurant owners to define their delivery zones by drawing them on a map. Our software will then store the region's boundaries in a database. Then, when a client requests delivery for an address, the system will automatically detect which zone that address falls onto, and, based on the zone's settings, will automatically inform the operator if the zone is unavailable for delivery, be it due to time restrictions or geographical restrictions (such as distance). It will also automatically apply surcharges, help inform delivery time calculations, among other things.

Figure 1: Google Maps drawing API example

1.3 Research Hypothesis

The proposal for this Masters' research hypothesis is the following:

"*By implementing a delivery solution that integrates directly with its Cloud management software, and considering the substantial number of clients Wintouch already has for its product, they will be able to provide a better service to its customers (both restaurants and end consumers alike).*"

1.4 Working Methodology

After completing the bulk of the research stage, work started on proof of concept versions of the project. On recommendation from both the company and my advisor, an Agile methodology was attempted as the main workflow. Early on, new iterations would take a few weeks to complete. However, as time went on, and the methods used became more refined, iteration times were reduced to about a week.

The weekly meetings were conducted with the presence of the company's senior leadership, and the Cloud project manager, where this dissertation project is included.

Considering all of this, the following iteration cycle was settled on:

1. Research for the subject and/or goals defined in the last weekly meeting;
2. Create a proof of concept or a mock-up, according to the research performed;
3. Implement the feature (if appropriate);

4. Request some feedback before the next full meeting, if necessary;

5. Prepare the next meeting.

At the end of each meeting, new objectives were defined for the next meeting, and a timeline was defined for them. These meetings were also an opportunity to propose new ideas, ask questions about the direction the project will take, and in general keep track of the progress and keep the project flowing smoothly, with everyone interested being regularly updated on its status.

Outlined here is the basic approach that was taken every week. However, the schedule was subject to change if necessary (depending on the availability of all parties, and whether or not enough time had elapsed to complete the objectives, or other factors that may have arrisen).

Also, the research steps mentioned above might not necessarily resemble traditional academic research (i.e. looking at academic journal articles). Most of the time, these research steps were conducted by inquiring other members of the company (as well as project leadership) about the currently implemented solution, clarifying their vision for the product, and researching competing products and possible solutions to the problems that arose.

1.5 Document Structure

This document's structure is as follows:

1. In this first chapter, an introduction to the thesis' theme has been given, by presenting the motivation for this project, the objectives of this project, as well as the research hypothesis and methodology that was followed during the development of this project.

2. In the second chapter, an overview of the market that this project is intended to serve is given, and an introduction to some important concepts is given.

3. In the third chapter, an analysis of the current State of the Art in this area will be presented.

4. In the fourth chapter, the proposed approach to solving the problem will be presented.

5. In the fifth chapter, the first functional prototypes developed for the project will be presented.

6. In the sixth chapter, some of the main challenges faced during the development of this project are presented, as well as the way that they were overcome is explained.

7. In the seventh chapter, a guided tour of the final version of the application developed is provided. Screenshots of that system will illustrate the various steps an employee shall go through to accept, process and deliver an external order.

8. In the eight chapter, after a summary of the dissertation's chapters, the final conclusions about the work done are discussed, comparing the real achievements against the original objectives and proposal. Also, some recommendations for future work are included.

Chapter 2

Background

This section aims to give the reader an introduction to the concepts required to understand this report. The objective is to give the reader a good understanding of specific terminology in the field, as well as the processes that exist, and make sure the reader understands them, so as to be able to read and understand the rest of this work.

2.1 Order Management system

This is the main software module that will be built. This software module will be responsible for, among other things, registering new requests and, in general, managing their state, since they arrive in the system until the order is finally delivered to a client. In the next few paragraphs, some of these concepts are clarified.

A delivery request, or an order, is a request by a client to have some food items delivered to a specific address, at a set time. This address does not need to be the client's home; it could be their work, a friend's house, or sometimes, although rarely, even their car. The delivery time also doesn't need to be "immediate". A lot of clients will request food earlier than they actually need it, in an attempt to receive it at just the right time, later on in the same day, or even a few days later.

Deliveries are handled by couriers. Couriers may be employees of the restaurants themselves, or may be employees of a third-party service. They may also not be employees at all: big delivery apps, such as UberEats, will hire couriers as independent contractors, due to fiscal advantages of not having so many employees of their payroll and not having to pay the associated benefits, thereby making them a more profitable company overall. The compensation schemes for couriers are outside the scope of this work. However, one of these schemes directly impacts this work: when the couriers are restaurant employees. In this case, this project will manage these couriers, attempting to handle all of the complexities of route

calculation, when to leave to arrive on time, which orders to take, etc.

Finally, a client is someone who, through some method, has placed an order with a restaurant.

2.2 Delivery request process / Ordering process

This subsection aims to explain the process of making a request from a restaurant. The traditional approach is through a phone call: a client calls a restaurant and requests food to be delivered at some time to some address. In them, at least a few pieces of information are required: the client's name, address, phone number, and the items they will be ordering. The operator, who answers the call, might also try to sell the client extra products, or the client might want to order something they've already ordered, but for one reason or another, doesn't know the name. For this reason, it is useful to have the client's order history easily browsable. After the client hangs up, the order is finalized in the system, and, if necessary (given the delivery time specified by the client), sent off for cooking.

More modern methods include the use of smartphone apps or websites. The process is different on these, as it doesn't require an operator on the other side. However, these are traditionally more expensive to setup, even if they're cheaper to maintain, and as such, a lot of restaurants in the Portuguese market still opt for the traditional method of using phone calls.

2.3 Order delivery process

This subsection will explain the process for when a courier delivers food to a client. This process starts when a food item is finally finished cooking. Traditionally, a courier will have some place where they can look at to see which deliveries need to be delivered. It is the intention of this project to implement a specific screen for this in the main software module that will be developed in with this thesis. However, it isn't required; manual methods are still used in many places.

Regardless of the method used, the courier will then select which deliveries he will take on his next run. He will print out the specific receipts (if necessary), mark them as being delivered, and physically load them for delivery.

Upon arrival at each of the courier's destinations, he will attempt to find the client. If he fails, he uses the phone number on the receipt to inform the client that the food has arrived, and ask for help with the delivery. The client will then accept the food, and, if necessary, pay for it, either with money, credit/debit card, or another method that is accepted by the courier. Then, the courier will move on to his next delivery, or will return to the restaurant to pick up new orders.

2.4 Wintouch Cloud Application Architecture

To close this Background chapter and because the new application to be built must follow the company's software devolpment approach and the new product shall interact with the existing applications, this section provides a brief overview of the Wintouch Cloud application's architecture.

The application is mainly composed of three parts:

1. The database server, which stores all of the application's data;
2. The Backend API (Application Programming Interface), built using .NET Framework 4.8[1], and using an older ORM (Object-Relational Mapper, in this case, Linq-to-SQL[2]) to communicate with the database;
3. The Frontend application, built using the Angular web framework[3].

These main services also interact with other external APIs and Services, such as the Azure Blob Storage Services[4] for storing images and other large files.

Both the Backend and Frontend applications are hosted on Azure App Services, which allows for easy deployment of the applications, and also provides a scalable, reliable, and secure hosting environment.

[1] https://en.wikipedia.org/wiki/.NET_Framework
[2] https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/sql/linq/
[3] https://angular.io/
[4] https://azure.microsoft.com/en-us/products/storage/blobs/

Chapter 3

State of the Art

As an initial note, it must be stated that research for this project initially began with the potential objective of exploring the Delivery app ecosystem. However, as the research progressed, it became clear this project's objectives would need to be adjusted, and the app idea was put on hold in favor of the approach outlined in this document. As such, this chapter is structured in the same way that the research progressed: first by exploring Delivery apps, analyzing the results of this research, and then, presenting the proposed solution.

Research for this project had the following objectives:

- Identify the current solution that Wintouch already has deployed in their Desktop line of products;
- Explore the current market landscape in Portugal;
- Locate shortcomings of their current product;
- Investigate existing solutions implemented in other markets, and analyze their potential application in the Portuguese market.

As per the company's request, a lot of attention was also given to the market analysis in the United States.

3.1 Analysis of the US Market

Research for this project began, as already stated, by identifying the main competitors in the US market of Delivery apps. This market is, as per Bloomberg's Second Measure, mostly occupied by three main

competitors: DoorDash, UberEats, and Grubhub[1]. As such, this first subsection will focus on analysing this market.

It is evident that, early in 2020, these apps saw massive growth, with Bloomberg citing growth of over 13% year-over-year in November 2021, due in large part to the rise in demand sparked by the COVID-19 pandemic. Average sales per customer jumped for nearly all apps, with DoorDash seeing the biggest increase out of the ones listed, with the money spent per customer nearly doubling from the first quarter of 2020 to the second quarter of 2021.

Over the last two years especially, and excluding the pandemic's influence, a number of events have occurred in the US market that have caused the main players in it to thrive: first, DoorDash went public[2] in December 2020, with its stock soaring 82% in the first day of trading alone, having been valued at nearly $16 billion USD just before the aforementioned IPO[3]. DoorDash has also made partnerships with CVS[4], Albertsons[5] and other regional and national convenience stores[6], and offering grocery deliveries through its app. In September 2021, DoorDash also announced that it was adding an alcohol delivery service as an option to its app[7].

Beyond that, UberEats also expanded its market share by acquiring Postmates, another meal delivery service present in the US market[8], and also in an attempt to diversity its business, UberEats also launched a U.S. grocery delivery service[9], and also partnered with Costco for same-day deliveries[10]. Uber also bought alcohol delivery company Drizly in the same time frame[11].

The results of these transactions and investments are clear: DoorDash stands at 59% of market share in April 2022, meanwhile UberEats came in second place with 24%, according to the same report by Bloomberg[12].

Another metric indicating the success of these investments is the average quarterly sales per customer. In this metric, DoorDash and UberEats have increased 89% and 72% between the first quarter of 2020 and the first quarter of 2022, respectively,

From the analysis performed, it also became apparent that one of the newest features introduced by the large U.S. apps were subscription services, attracting large shares of customers in all major services provided in the United States.

[1] https://secondmeasure.com/datapoints/food-delivery-services-grubhub-uber-eats-doordash-postmates/
[2] https://www.cnn.com/2020/12/09/tech/doordash-ipo/index.html
[3] https://www.cnn.com/2020/06/18/tech/doordash-funding-valuation/index.html
[4] https://www.supermarketnews.com/online-retail/cvs-taps-doordash-same-day-delivery-groceries-and-non-rx-items
[5] https://www.cnbc.com/2021/06/21/doordash-and-albertsons-partner-on-same-day-grocery-delivery.html
[6] https://doordash.news/2020/04/01/bringing-household-essentials-to-your-doorstep-offering-convenience-beyond-food/
[7] https://www.bloomberg.com/news/articles/2021-09-20/doordash-launches-alcohol-service-stepping-up-delivery-wars
[8] https://www.restaurantbusinessonline.com/technology/uber-eats-completes-postmates-acquisition
[9] https://www.grocerydive.com/news/uber-scales-up-grocery-business-as-delivery-operations-overtake-ride-sharin/588543/
[10] https://www.retaildive.com/news/costco-pilots-same-day-delivery-with-uber/603773/
[11] https://apnews.com/article/business-1763524b276ddef6b7f247753d356d6f
[12] https://secondmeasure.com/datapoints/food-delivery-services-grubhub-uber-eats-doordash-postmates/

3.1.1 Common themes

A few themes that pretty much all apps have in common, according to the research done, are:

- Easy to use interface;
- Powerful search features;
- Delayed requests (i.e., order now, deliver later);
- Convenient payments;
- Subscription models;
- Group orders.

3.1.2 Profit maximization strategies

In this section, some of the strategies that these apps use in order to maximize their profits are explored. Two of them have already been mentioned:

1. Consolidating their market share with horizontal integration (i.e., buying smaller competitors);
2. Partnering with local and national chains to offer new items for delivery, such as alcoholic beverages and everyday groceries.

However, there are two more strategies that most apps regularly use:

1. Order packing: simply maximizing the number of orders delivered in each delivery run, by optimizing the internal algorithms used to determine the routes couriers will take on each run;
2. User tracking: it is commonplace for apps to track the habits and preferences of their users, both to improve their own business and also to sell that data to advertisers.

3.1.2.1 Order packing

The problem of order packing, in the area of food delivery, seems to be simpler than its more general version of last-mile Delivery optimization, because of one major difference: the problem of missed deliveries is much smaller in the food delivery domain, than it is for the general problem, because it is presumed that, when a customer requests food, they will still be available later when the food needs to be delivered at their location, unlike the general problem, where customers might order a package that could arrive days, or even weeks later [1]. As such, this problem simplifies down to a problem of optimizing the delivery routes according to the orders the courier must deliver, minimizing delivery times for the individual orders, even if it means sending more than one courier to the same location, otherwise, warm food items might, for example, become cold, if they wait too long to be delivered because of the delivery of other orders.

3.1.2.2 User tracking

It is a fairly common practice across the industry that software companies will track information about their users [3]. The analyzed delivery apps are no exception: DoorDash's privacy policy specifically states that the service collects information about you, such as your name, email address, delivery address, phone number, and other information that is required for the service to work[13,14], including the items you order, together with any special instructions and payment method used. Given that this data is required for use in every other delivery service, it is assumed that these services will also collect this data.

DoorDash also uses other technologies to track the user experience inside their app. Logging in with third party services, such as Facebook or Google, will also share that information with those services, adding to these services' abilities to track users and user habits across the web and in the real world.

There is a legitimate need for all of this data: these apps need your name, location, order items, and payment information to both place the order and deliver the food to the customers. However, all of this data is then combined with data from other sources (such as advertising networks) for many other purposes, including, but not limited to, customizing the user experience, recommend similar restaurants in the future, and, of course, deliver targeted advertising. There are obvious privacy concerns with these practices. However, as this was not the topic of this work, they weren't explored in great depth.

All of these conclusions were presented to the company, and eventually it was decided that implementing these collection features would be a large enlargement of the scope of work to be done, and therefore, it was decided they would not be implemented.

3.1.3 Other relevant features

During the research into the US market, other innovative features that are being used in the industry were also explored. In this section, some of them will be explored, their potential application to this project, and ultimately which ones the company decided to move forward on.

3.1.3.1 Alternative order placement channels

One of the most interesting features explored was the ability to place orders through other channels, such as Amazon Alexa, Google Assistant, Facebook Messenger, WhatsApp, or others. This feature is particularly interesting because it allows the user to place an order without having to open or install an external app, and using these apps' bot features, it is possible to build a completely customizable and convenient experience for the user to order food and other items.

[13] https://blog.avast.com/what-food-delivery-apps-track-avast
[14] https://help.doordash.com/consumers/s/privacy-policy-us?language=en_US

3.1.3.2 New delivery methods

Another interesting feature explored was the ability to deliver food through drones. This feature is particularly interesting because it promises to deliver food faster and more cheaply than traditional delivery methods. However, this typically requires enourmous upfront capital investments, and the technology, while progressing, is still in its infancy, so it is not a viable option for the company at this time.

3.1.4 Chinese Market Analysis

In this section, an analysis of the Chinese market will be performed, because of some articles that were explored for the research presented in the previous sections comparing it to the US market. It will be a much smaller analysis, focusing on one particular feature that was suggested to the company for this project.

First, some background: using Dauxe Consulting data we can see that the Chinese market is dominated by Meituan, with 70% of the market share, followed by Ele.me, with 26%. The remaining is shared by other smaller players[15]. Both the US and Chinese markets are highly consolidated, with a few large companies controlling the vast majority of both markets.

However, one of the better features that was explored during the research into the United States market was actually a very popular feature in China: discount coupons given out by the restaurants themselves, mainly through local apps such as WeChat[16]. Because of its extreme popularity in China, it was suggested to the company that this feature be implemented in the app, as a way to attract more customers.

3.2 Analysis of Wintouch's current Desktop solution

The Wintouch Desktop solution (Desktop application, Desktop version) has been in the market for 20 years, and its Delivery module also has over a decade of market experience. However, given its age, it does a lot of things differently from what is possible to build today. Regardless, it is still a very powerful tool, and it is still used by many of Wintouch's customers.

As such, an analysis of the Desktop solution was also made, in order to understand its strengths and weaknesses, and to propose to the company new ways in which it could be improved. Those proposals, and the ones that were accepted, will be discussed in Chapter 4 - Proposed Approach.

3.2.1 Basic Arquitecture

The Wintouch Desktop application is an on-premises business management solution, which means that it is installed on the customer's own servers, and is accessed through terminals that have the required

[15] https://daxueconsulting.com/o2o-food-delivery-market-in-china/
[16] https://radii.co/article/food-delivery-china

Desktop applications installed. It is a multi-module solution, which means that it has many different modules, each one with its own purpose. The Delivery module is one of these modules, and it is the one will be focused on.

The Delivery module is integrated into the Restaurant module (*Restauração*), which is the module that handles the restaurant's operations. The Restaurant module is the one that handles the restaurant's menu, and the Delivery module is the one that handles the delivery of the restaurant's orders, accessible through the Restaurant module, or independently accessible on some posts, if configured by the restaurant.

3.2.2 Delivery module

The Delivery module itself contains a fairly comprehensive set of features, such as:

- Order management: the module allows the restaurant to manage its orders, including the ability to create new orders, edit existing orders, and cancel orders.
- Delivery zone management: the module allows the restaurant to manage its delivery zones, including the ability to create new delivery zones, edit existing delivery zones, and delete delivery zones. Zones can be created by adding them onto the right table (with the right editor), and once inside the creation forms, you can then manually add all of the addresses that should be part of the provided zone. You can also configure parameters for each, such as a delivery fee, or the working hours for the zone (that is, the hours during which the zone is active and couriers will deliver food to addresses in that zone).
- Delivery order type management: the module allows you to create new types of delivery orders, based on the two basic "classes" of orders: "Delivery" and "Takeaway".

Given that these features already exist, these features were demanded by the company for the Cloud version. A very clear improvement over the Desktop solution, however, will be the new system for Delivery zones.

Chapter 4

Proposed Approach

This Master's project is oriented towards, as already stated, primary objectives as well as secondary objectives. During the course of the first few weekly meetings with the company, the objectives and scope of the project saw rapid changes, based on the research conducted and discussed in the previous chapters, as well as the company's objectives.

The current main objectives boil down to fully implementing the software components, depicted in the block diagram of Figure 2, such that a client can successfully place and order and have it delivered by a courier, with all the necessary features and functionality implemented for such a task.

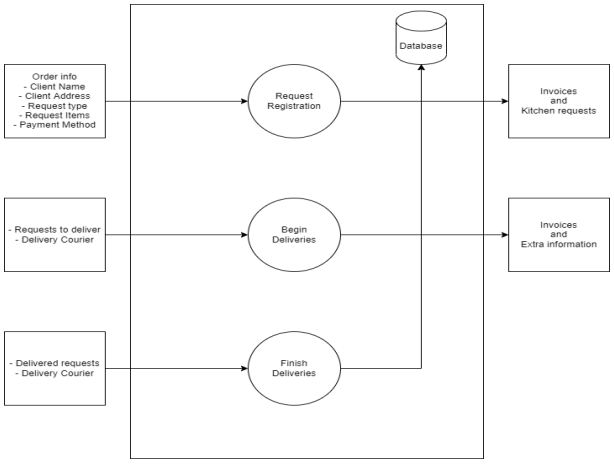


Figure 2: System Architecture

The four main software components that compose the system architecture shown in Figure 2, are described below.

- Request registration - This component takes in input from the user, specifically the request details (such as the client's name, address, whether the request is a delivery request or a pickup request, and other necessary information), and registers it to the database (calculating any and all necessary information that may need to be calculated beforehand), and also physically prints out invoices (if necessary) or kitchen requests (that is, a piece of paper that is sent to the kitchen and serves the purpose of telling the restaurant staff what to cook);

- Begin Deliveries - This component is responsible for taking a courier's input, specifically, by taking in the requests the courier will deliver, and mark them for delivery, and also, it will print out the invoices again (if required);

- Finish Deliveries - This component will, upon a courier's request, mark the courier's deliveries as finished. It is responsible for adjusting payment methods, if needed, and ensuring that the client's money is accounted for. This module also must handle exceptional delivery cases, such as if the order is returned.

- Delivery zone mapping - This component will allow users to properly configure the delivery zones, using features such as the ability to specify which addresses belong to which zones, and also, it will allow the user to configure the delivery fees and other parameters for each zone.

All of these modules have detailed implementation flowcharts developed, that received feedback from the people responsible for the project inside the company during their development.

4.1 Early mock-ups

In this section, the first mock-ups that were created for this project are presented. Most of them were rejected by the company. Explanations for these rejections (and the suggestions for improvement) are given below.

The first mockup, shown in Figure 3, was intended to be the landing page for the Delivery module. This page was rejected for being too cumbersome; showing too much information, and not being required for operators who only require access to registering new requests to access.

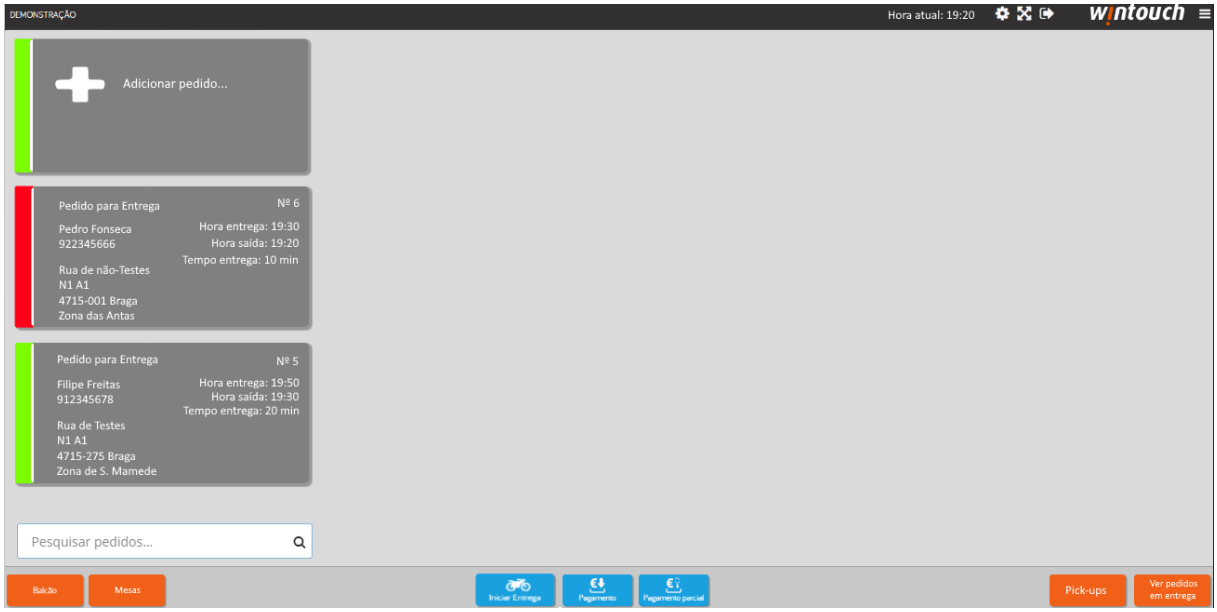


Figure 3: Landing page

The next mockup, shown in Figure 4, would show up when you clicked the button, on the previous page, to add a new request that just came in to the restaurant. This screen actually remains in the latest versions; however, instead of being in a modal, it is on a stand alone page. It allows an operator to search the client database for clients, and allows an operator to add a new client if necessary.

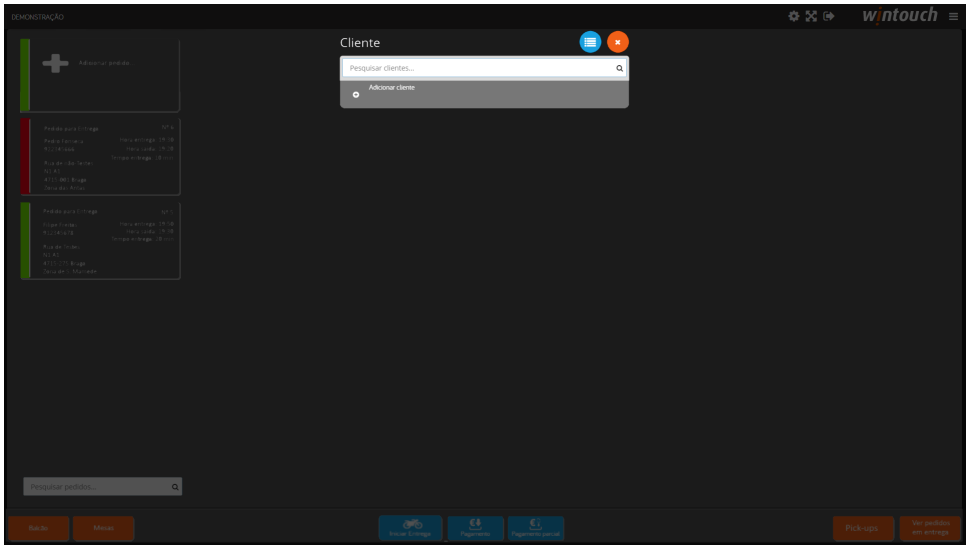


Figure 4: Client search page

The following mockup, in Figure 5, allows an operator to type in information related to the delivery request, such as the client's name, address, telephone number, and other required information. It remains in the final version, however, it does contain some changes.

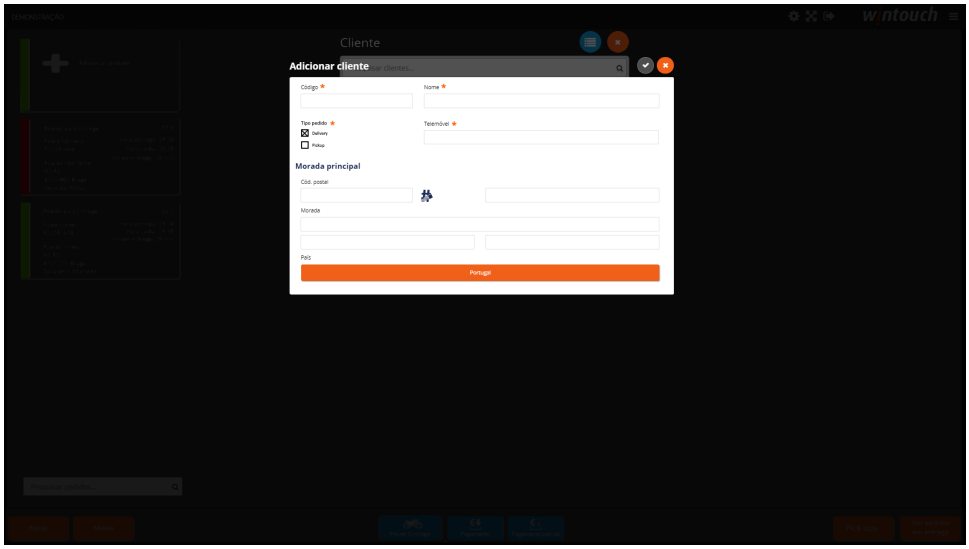


Figure 5: Delivery Request Details Page

The final mockup, in Figure 6, would show right after the previous one, and is where an operator will insert the products requested by a client. It too remains, however with some changes, which will be made obvious later on.

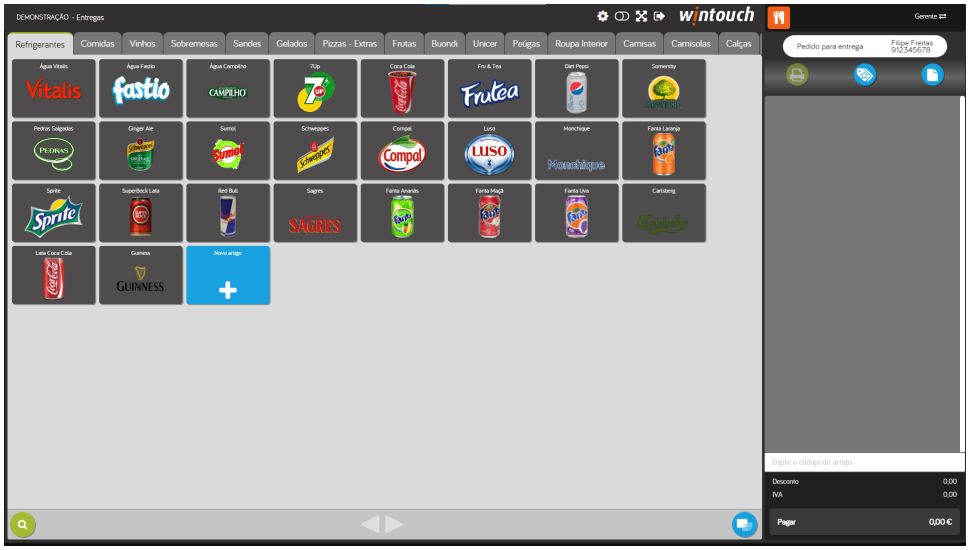


Figure 6: Keyboards Page

These mockups, while were rejected in their majority, did lead to the creation of the first functional prototypes. These prototypes were used to test the system, and to gather feedback from the company, which was then used to improve the system.

Chapter 5

First Functional prototypes

In this chapter, more screenshots will be presented. These screenshots are of the first version of the prototypes that were built.

The first screen, shown in Figure 7, is equivalent to one of the previous screens, showing how an operator may search for clients.

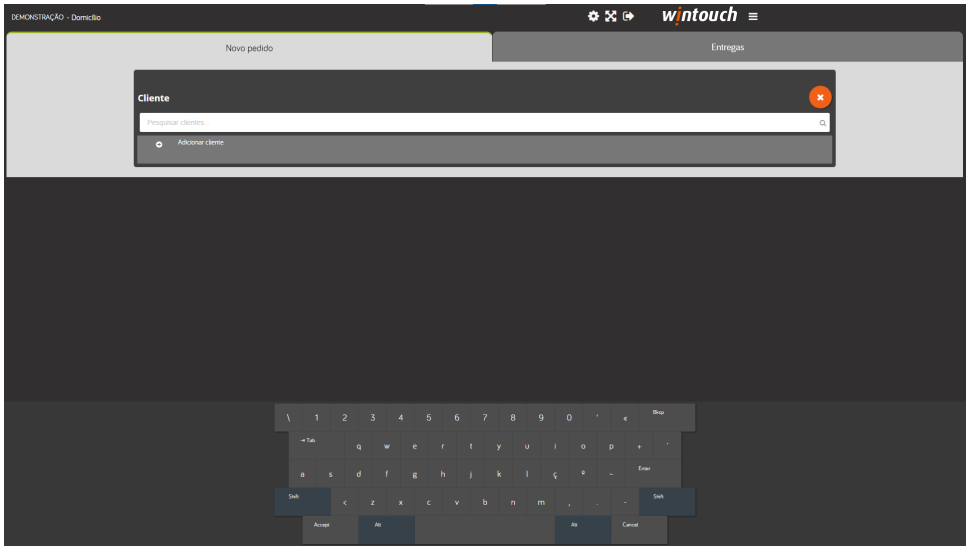


Figure 7: Client Search page

The next screen, shown in Figure 8, is an improved version of a previous screen. It shows the form for inputting the delivery request details, such as client name, phone number, and address.

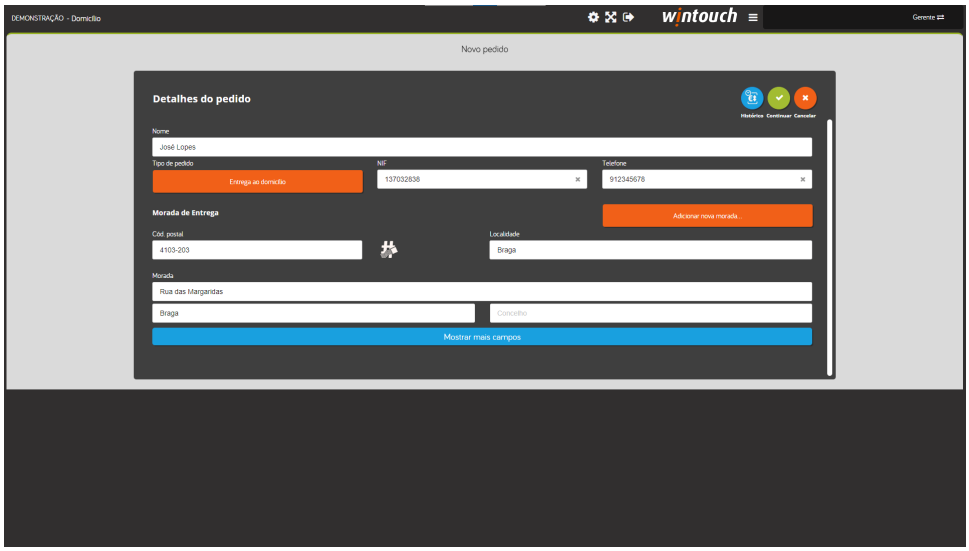


Figure 8: Delivery Request Details page

The mockup shown in Figure 9 is for a new screen: it contains an history of the client, if applicable (it doesn't show up on new clients). Most of these statistics are automatically computed already, however, some of them are added only for demonstration purposes.

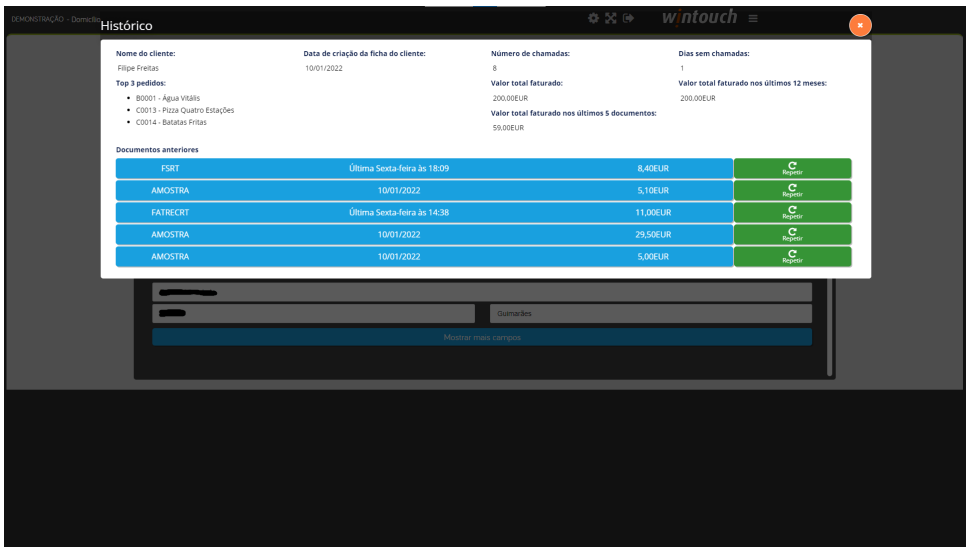


Figure 9: Client History page

Last but not least, the last screen, still incomplete, in Figure 10, allows an operator to insert the client's requests into the system.

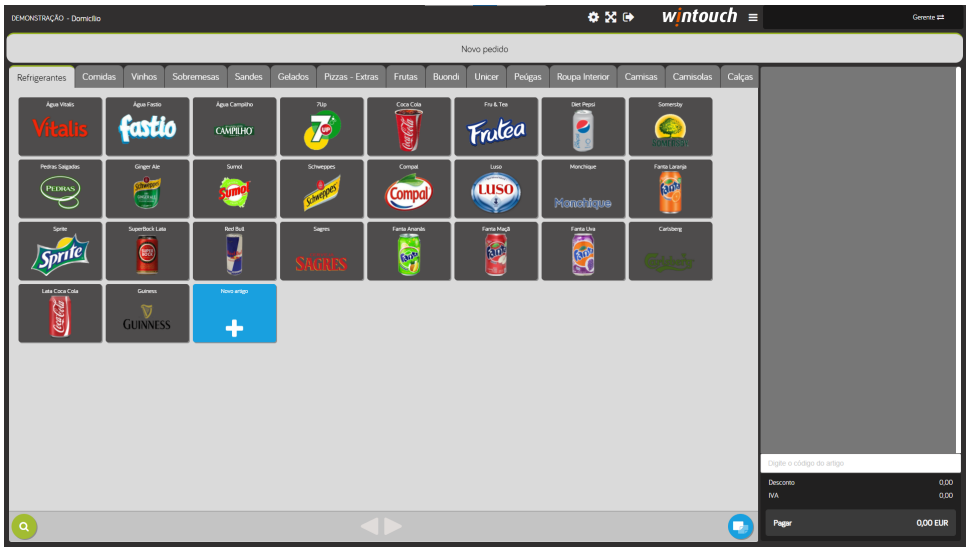


Figure 10: Keyboards page

5.1 Bugfixing iterations

During my time at the company, due to unforeseen internal issues that arose, I was unable to work on my dissertation project for a period of a few months, instead being involved in helping the team fix a lot of issues that were showing up in the Cloud application. This was a very important experience for me, as it allowed me to learn a lot about the application, and once I came back, I was able to significantly speed up my development work on my own project.

This experience taught me a lot about the architecture of the application, having learned a lot about the internal processes of the API, the database structure, as well as the more obscure parts of the frontend application. I also learned a lot about the way the company works, and how the different teams interact with each other. This experience was later put to use when I finally came back to work on my own project, as I was able to quickly identify and fix a lot of issues that were showing up in the Delivery module, as well as properly make use of all of the design patterns and specific functionality built for the rest of the Wintouch Cloud application, avoiding a lot of bugs that could occurr otherwise.

Chapter 6

Main development challenges

As mentioned before, the development of this project went through several iterations, with weekly meetings with company leadership, and during each meeting, progress from the previous weekly meeting was presented, discussed, and feedback was provided. As problems arose, or if the company was not happy with the direction of the project, changes were immediately made.

However, during the development of the project, as is predictable, many problems arose that had to be solved. This chapter intends to provide an overview of those problems, as well as the approaches presented to solve them, and the solutions that were eventually decided on during the various meetings.

6.1 Development of the final versions of the mockup screens

Many weekly meetings were spent simply trying to finish all mockups required for all of the user-facing screens that the application will have. Some of the most important iterations during the development of these mockups were the development of the deliveries monitor. The first set of mockups for the deliveries monitor screens are shown in Figures 11 to 18.

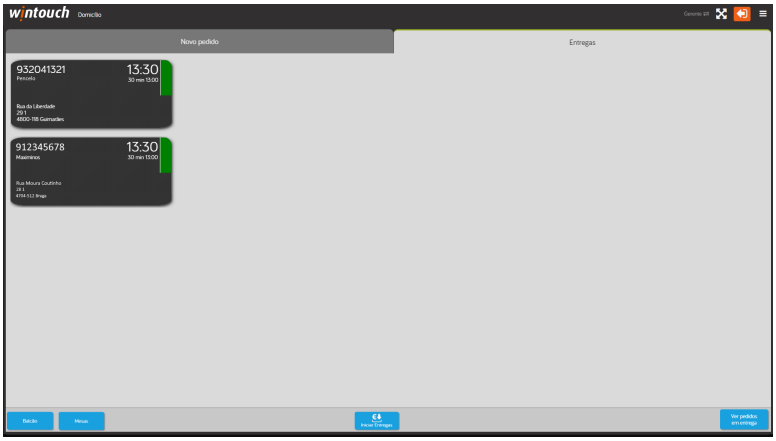


Figure 11: Deliveries Monitor - 1 - Main page

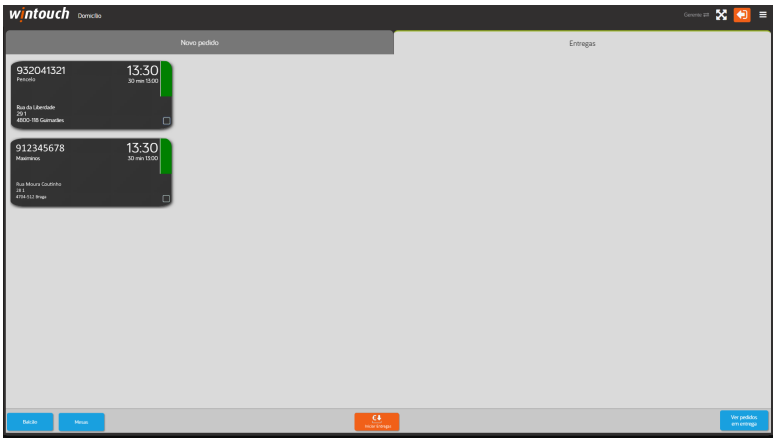


Figure 12: Deliveries Monitor - 2 - Select orders to start delivery

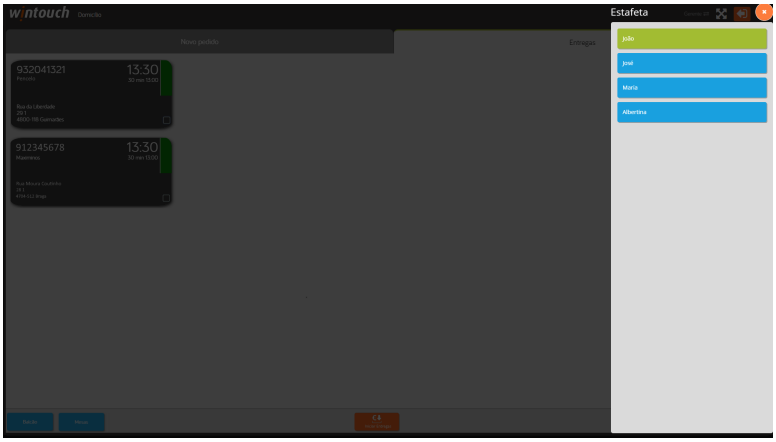


Figure 13: Deliveries Monitor - 3 - Select Courier

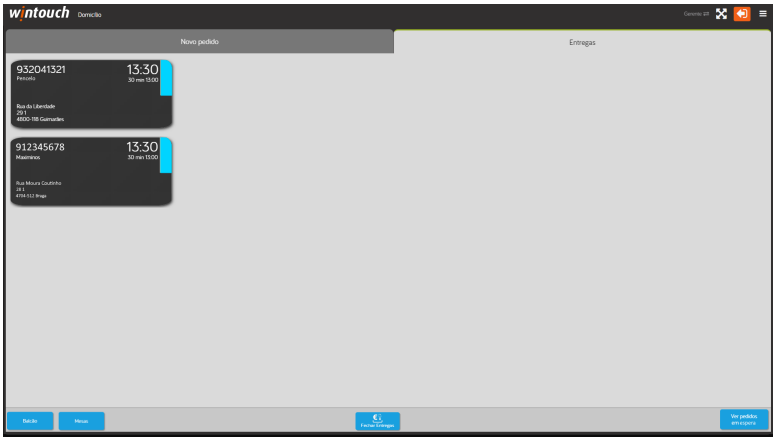


Figure 14: Deliveries Monitor - 4 - Orders in delivery

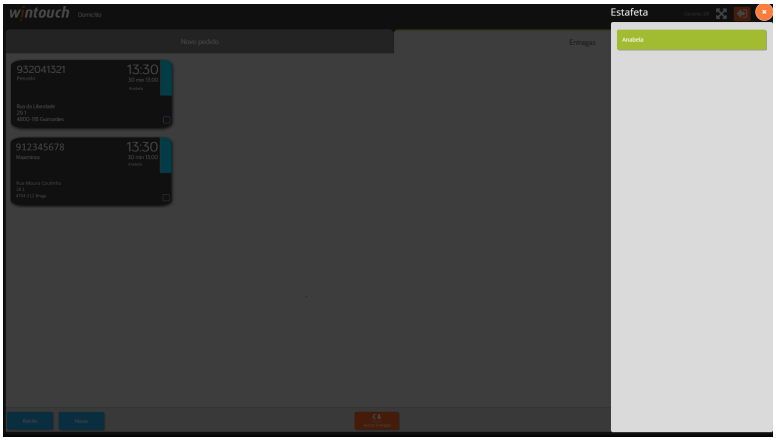


Figure 15: Deliveries Monitor - 5 - Select orders to close delivery

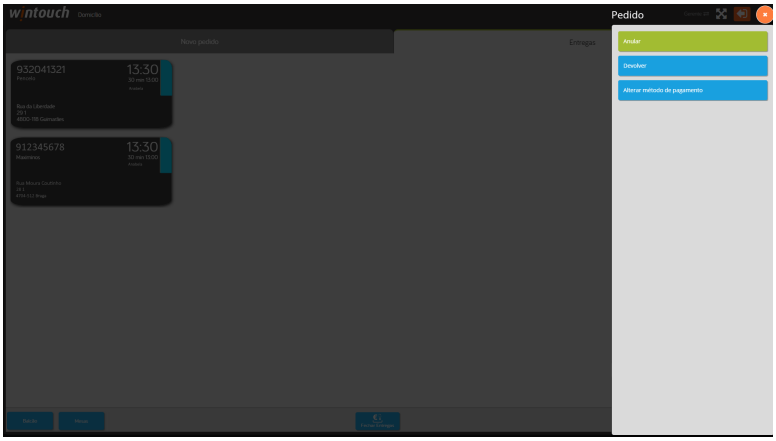


Figure 16: Deliveries Monitor - 6 - Manage order - Actions

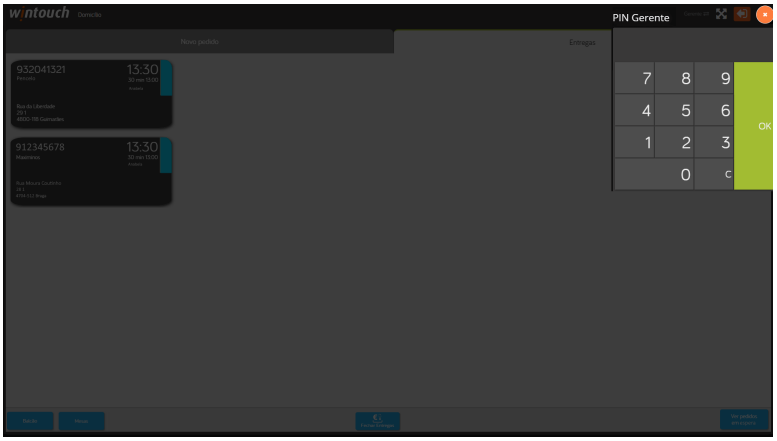


Figure 17: Deliveries Monitor - 7 - Cancel order - Manager PIN

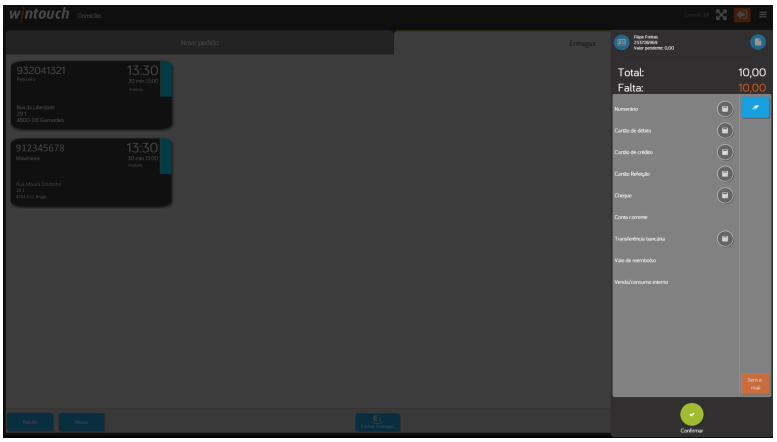


Figure 18: Deliveries Monitor - 8 - Change payment method

The screenshots in Figures 11 to 18 show all of the screens that were presented at one of the first meetings. However, during the next few weekly meetings, these screens were perfected, until the mockups in Figures 19 to 25 were eventually agreed upon.

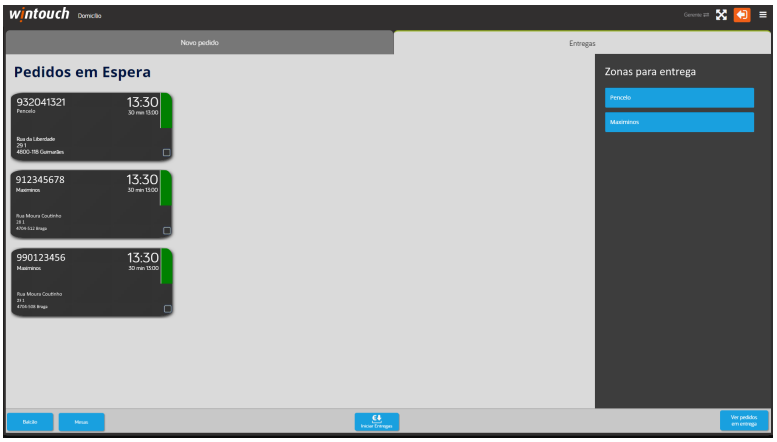


Figure 19: Deliveries Monitor - 1 - Main page

As we can see in Figure 19, the main changes to this page is that a new filter was added: the zones

filter. Usually, when a driver is going out to deliver requests, they will want to deliver multiple orders that are generally in the same location. As such, adding this type of filter is an important feature to have. Also, it is also allowed for a driver to select a zone, and then select other orders that are not inside of that zone, if they know that they will be able to make that delivery as well as the others. Along the same line of thinking, but in reverse, it must be stated that de-selecting a request inside of the selected zone is also allowed.

Another change that was made after the presentation of these final mockups was that a new tab should be added globally: the "Deliveries in progress" tab, which will replace the button on the bottom-right corner that previously allowed a driver to access the list of deliveries in progress. No mockups were made with this design, but, that is the design that is being used in the final product.

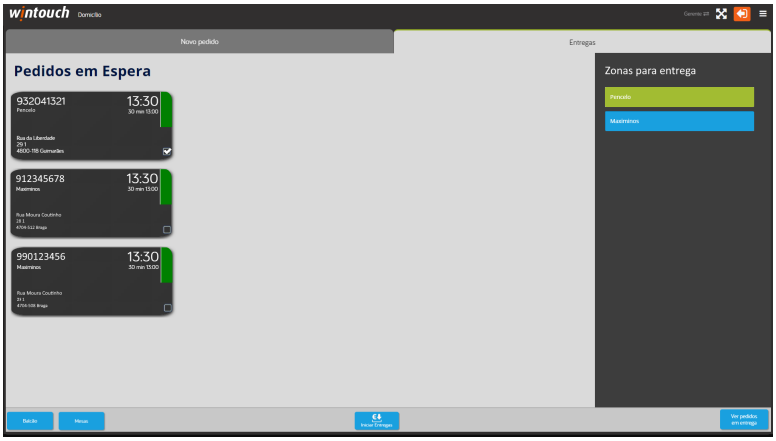


Figure 20: Deliveries Monitor - 2 - Select orders to start delivery

The page in Figure 20 simply shows what happens when a zone is selected, which is something that wasn't visible in the previous mockups.

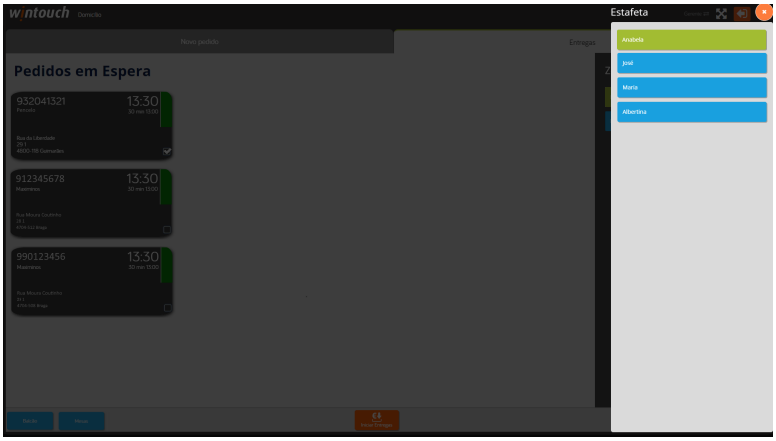


Figure 21: Deliveries Monitor - 3 - Select Courier from list

Upon clicking the "Start deliveries" button (on the bottom-center of Figure 21), the user will then be presented with a list of the drivers that are available to make the delivery, as seen in Figure 20. A driver

is considered to be available for deliveries if they are configured, by the manager of the store, as a driver for the purposes of the Delivery module, and if they aren't already out making other deliveries.

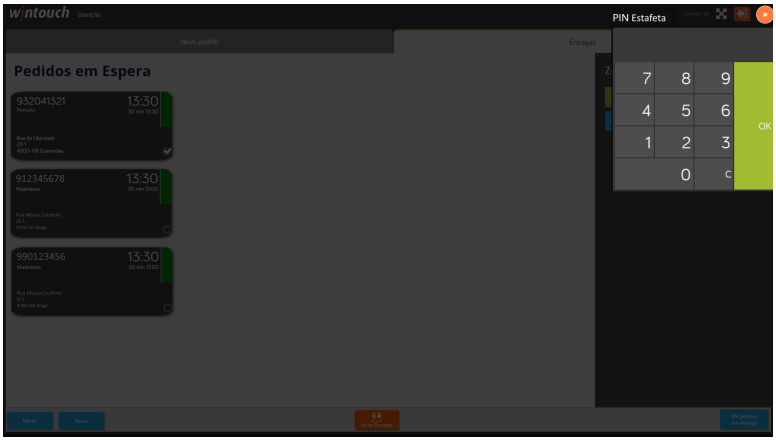


Figure 22: Deliveries Monitor - 3.1 - Select Courier with PIN

In the page in Figure 22, it can be seen that the driver has to enter their PIN in order to start deliveries. This mode is optional (configurable for the entire company), and, while a lesser used feature, it is useful in preventing possible cases of fraud.

Regardless of which mode is currently active, if there is only one driver available, then the system will automatically select that driver, no PIN or list required.

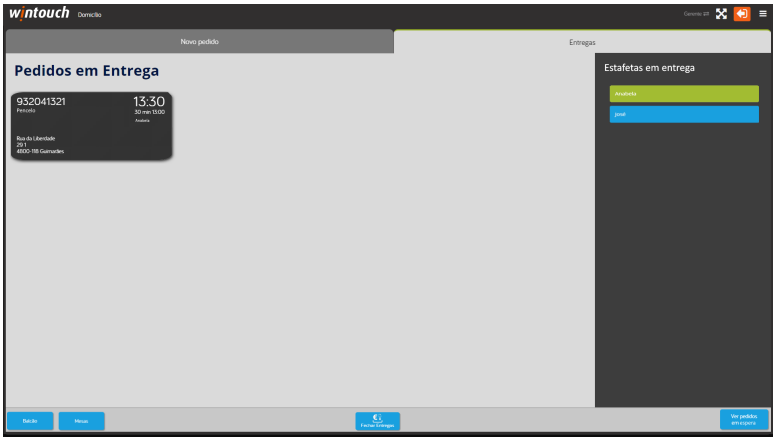


Figure 23: Deliveries Monitor - 4 - Orders in delivery

The page in Figure 23 now shows the list of requests that are currently being delivered by the selected driver. This page is, in these mockups, accessible from the bottom-right corner button, but, as mentioned before, this button will be replaced by a tab in the final product. A driver, when returning from their deliveries, will access this page to mark, in the system, their deliveries as being finished. They will then be redirected automatically to the previous page, where they will be able to start new deliveries. Alternatively, the driver may access special options about each request by simply clicking on it. A menu with available options will then be shown, as seen in Figure 24.

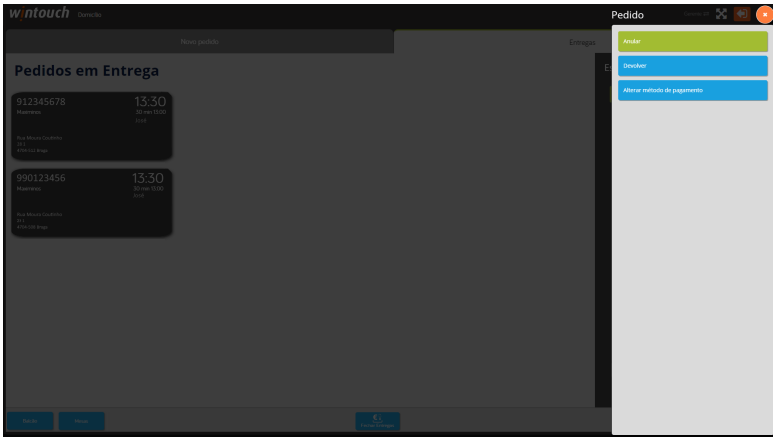


Figure 24: Deliveries Monitor - 5 - Manage order - Actions

In this modal (Figure 24), the driver has pressed one of the requests, and they can now select one of the special actions associated with the request. These actions are:

1. Return the request - this action allows the driver to return the request to the "Pending delivery" state, meaning it can be delivered again. This option is useful for situations such as the driver not being able to deliver the request, or the client not being at home, etc;

2. Cancel the request - this action allows the driver to simply delete the request from the system. This option is useful for situations such as the client not wanting the request anymore, or the client not being able to pay, or other similar situations. Given the severity of this option, it always requires that a store manager approve it with their own PIN;

3. Change payment method - this option allows the driver to alter the payment method that was stored with the request document. This option exists such that the accounting made by the application is always correct, and that the driver can change the payment method if the client decides to pay in a different way than what was originally planned. After the change is complete, the delivery is immediately closed as well, and, if the driver has no more pending requests, they are immediately returned to the previous screen. Otherwise, they stay in the same screen, allowing them to perform more actions on the remaining requests, or just close them.

If the driver wants to change the order's payment method, the screen in Figure 25 is the screen that will allow them to do so.

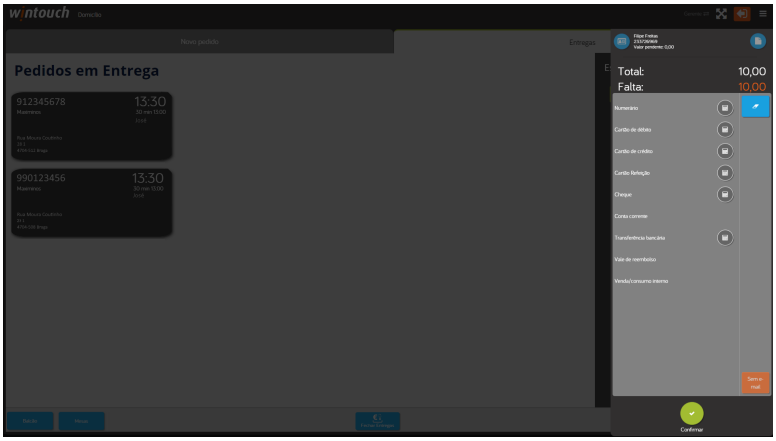


Figure 25: Deliveries Monitor - 6 - Manage order - Change payment method

6.2 Database tables design & creation of new data structures

The initial plan for the development of this project was to add a minimal amount of new fields to the existing database tables, especially the tables holding invoicing and associated information, given that those tables were already extremely complex in their design. However, that complex design would, at least it initially seemed, allow for the implementation all of the required functionality without problem. Regardless, it soon became aparent that a new data structure would be required to save all of the delivery request information that did not make sense in the document structure.

The Wintouch Cloud product already contains a data structure, a Product Document, that is used to store all of the information about a sale, including nested structures that hold the details of the sale's products. Foreign key associations from this table to other tables are also present, and used to link the sale to other tables, such as the client who made a purchase, the employee who made the sale, and other metadata that the application's user doesn't directly see, but that is used by the application for internal processing. One of fields present in this table, however, is its Source App Code, which indicates from which part of the application the sale was made (if from the Back Office, from the Point of Sale Counter, or from the Point of Sale's Tables module). For the Delivery module, a new possible value for this enumeration was added, indicating that the sale was made from the Delivery module.

Even though a decision to add a separate Delivery Request structure was made, a Delivery Request still has an associated document; the document associated with a Delivery Request is stored as a foreign key in the new table created to store the request's extra information. This approach was selected as a natural consequence of not duplicating all of the logic already existing for the Document, as the Document structure already contains information such as the order's contents (i.e., the products ordered), and the frontend components are already capable of dealing with adding new products to the order, paying a document, and doing all of the other required operations for the Delivery module. However, the Delivery Request structure is not a document, and therefore, a new saving mechanism capable of saving both of

these structures at once was added. This mechanism will also be discussed in this chapter.

As such, after many iterations, the following new tables were created with the structure described in the next three subsections.

6.2.1 Delivery Requests table

The Delivery Requests table, and the associated data structures, will be responsible for saving, among other things, the following information:

- The delivery request's ID, which is a unique identifier for the request;
- The ID of the type of order (i.e. pickup vs delivery);
- The ID of the client associated with this request;
- The name of the client associated with this request;
- Multiple fields that are used to store the delivery address;
- A field to store the address' delivery zone;
- The ID of the employee who took the client's request;
- The time at which the request was taken;
- The time for which the client wants the request delivered;
- The ID of the document associated with this request;
- The ID of the employee who is delivering the request (NULL if no driver has yet been assigned, or if the request is of type pickup);
- The start time of the delivery (NULL if the delivery hasn't started yet);
- The end time of the delivery (NULL if the delivery hasn't ended yet).
- Other metadata fields required by the application.

For every column in this table that stores an ID, the appropriate foreign key relationships have also been created.

6.2.2 Delivery Order Type

The Delivery Order Type table stores the different types of orders that can be made. An order can be a pickup or a delivery. However, for myriad reasons, restaurants may choose to create multiple types of orders for the same "class" of order (for instance, having two types of orders that are both pickups, as well as two types of orders that are both delivery).

This table's schema is pretty simple, containing just a single foreign key relationship:

- The ID of the type of order;
- The code of the type of order;
- The name of the type of order;
- The class of the type of order;
- The additional cost for this type of order;
- The product ID of the product that will be used to add fees to the invoice;
- Other metadata fields required by the application.

6.2.3 Delivery Zone

The Delivery zone table is actually not new. The Cloud application already contained a zones table, and it was decided to simply add new columns to it. This table is already used by the Cloud application for filtering purposes; there are already a lot of "explorations" in the application that allow you to filter by zone (for instance, total sales per zone). As such, new columns were simply added to support the requirements of the delivery module, and at the same time, because we're reutilizing the zones table, it means that all of the explorations already built for zones will continue to work, without any changes, with the Delivery module's zones.

Therefore, the following columns were added to the table:

- The ID of the zone;
- The average time it takes to deliver requests to this zone (in minutes);
- The surcharge for delivering requests to this zone;
- The start of the zone's delivery time window;
- The end of the zone's delivery time window.

There are no new foreign key relationships in this table.

6.2.4 Delivery Zone Address

The Delivery zone addresses table is new. According to the defined approach for delivery zones, this table will essentially map the IDs of the zones to the IDs of the addresses that are part of that zone. This table's schema is as follows:

- The ID of the address in the global address table;
- The ID of the zone this address belongs to;
- Other metadata fields required by the application.

The only foreign key relationship that exists in this table is the one between the zone ID and the zone table. There cannot be a foreign key relationship between the address ID and the global address table, as the global address table will not be in the same database as this table, and there is no guarantee that both databases will be stored in the same SQL Server instance. If both databases were in the same server, it would be possible to establish this relationship; but since that isn't guaranteed, it is impossible to create a foreign key relationship here.

6.3 Integration of the first screens

After the creation of the tables, the functional prototypes described above were then improved to integrate the changes and new entities created as a result of the new tables. As such, those screens evolved from mere prototypes to the first functional screens that included persistence of the data.

Some screenshots of these screens can be seen in Figures 26 to 29.

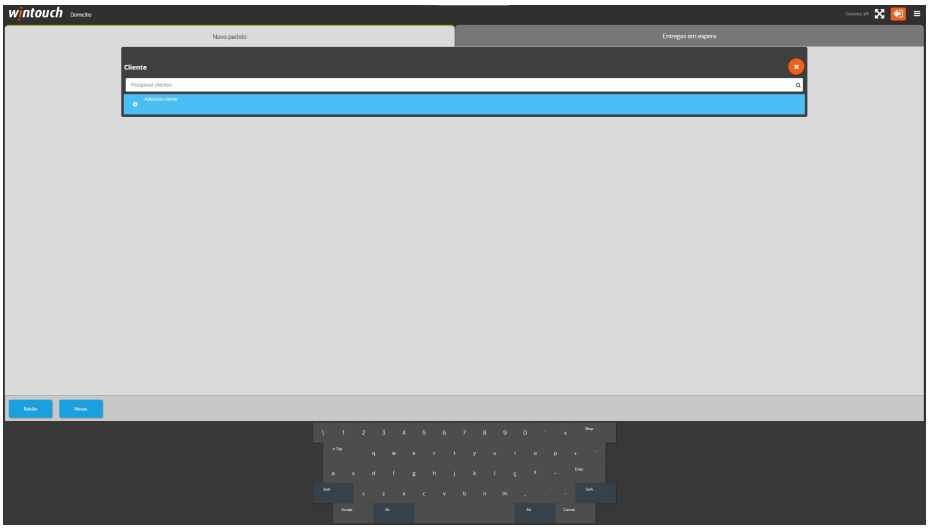


Figure 26: First functional screens - Main page

The screens in igures 26 to 29, in the order that they were presented, also show the entire flow of how to register a new request in the system. The screens are optimized to be easy to use on a touch

Figure 27: First functional screens - Request details page

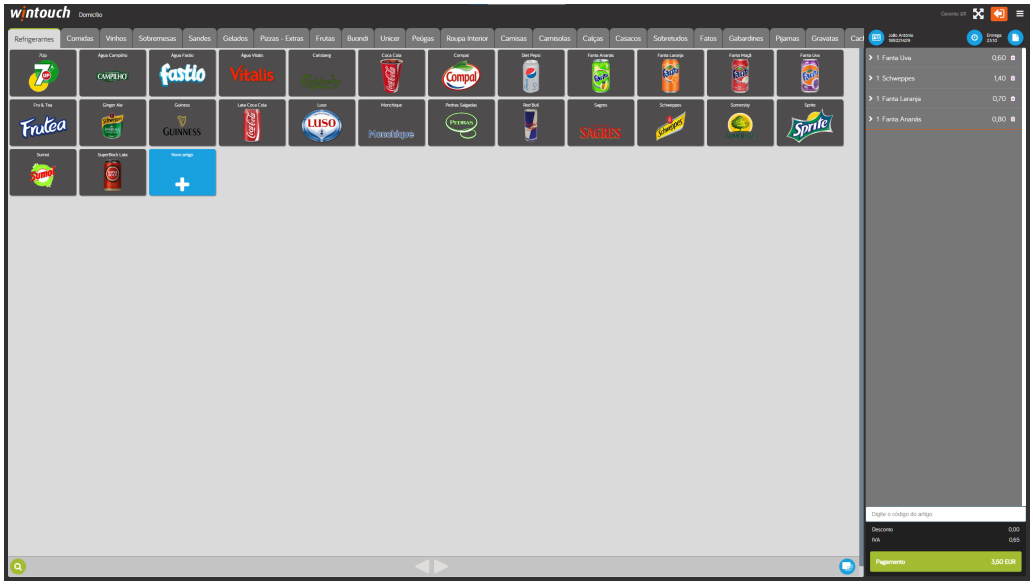


Figure 28: First functional screens - Request products page

screen, and also, they are optimized for a resolution of 1024x768. There is also another screen that was made fully functional in this iteration, which is the History screen. However, that screen has already been shown above, and the only changes made in this iteration were the automatic calculation of all of the statistics that it shows, and therefore, it doesn't make sense to show it again.

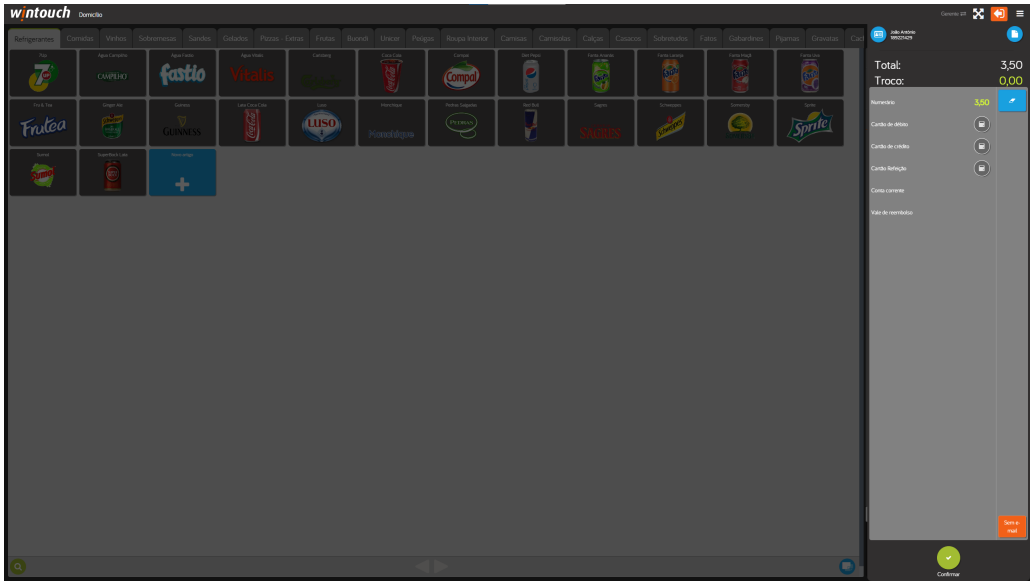


Figure 29: First functional screens - Request payment

6.4 Delivery-module specific Employee permissions

In the weekly meetings, it was decided that the delivery module should have its own set of permissions, and that these permissions should be different from the permissions that the Cloud application already has for the different POS modules. As such, the following permissions were selected for the Delivery module:

6.4.1 Document permissions

1. Set discount - allows an employee to set a global discount on the entire order;
2. Set VAT included - allows an employee to change the VAT included status of the order. This setting specifies if the VAT is already included in each line of the request, or if VAT must be added to each line of the request;
3. Price line - allows an employee to change which price line will be used for new products added to a request;
4. Market - allows an employee to change the market of a request. The market can be National, EU Internal Market (*Intracomunitário*, in Portuguese), or Others;
5. Printing model - allows an employee to change the printing model of a request. The printing model can be a ticket receipt, or some other type pre-configured by the user or included by default in the application;
6. Document observations - allows an employee to add observations to a request;

6.4.2 Line permissions

1. Warehouse - allows an employee to change the warehouse of a request line;
2. Free comment - allows an employee to add a free comment to a request line;
3. Discount - allows an employee to set a discount on a request line;
4. Description - allows an employee to change the description of a request line;
5. Price - allows an employee to change the price of a request line;
6. Quantity - allows an employee to change the quantity of a request line;
7. VAT Tax - allows an employee to change the VAT tax of a request line;
8. Total - allows an employee to view (but not edit) the total of a request line;
9. Unit - allows an employee to change the unit of a request line;

6.4.3 Special operations

1. Delete requests - allows an employee to delete requests;
2. Delete requests after invoice - allows an employee to delete requests after they have been invoiced;
3. Insert products by code - allows an employee to insert products by code;
4. Product offer - allows an employee to offer a product in a request;
5. Older documents - allows an employee to view older documents/requests from any client;
6. List products - allows an employee to view the list of products;
7. Edit clients - allows an employee to edit a client's information;
8. Open smart drawer - allows an employee to open the smart drawer;
9. Exchange money smart drawer - allows an employee to exchange money in the smart drawer;

These permissions were all added to the backend in the same way the other permissions already existed. Because of that, it means that these are loaded on the frontend together with the other permissions, and as such, as the new screens were developed, these permissions were immediately respected by the application.

6.5 Changes to client history screen

As mentioned in the previous section, the History screen was already fully functional, but it was decided, during the weekly meetings, that it should be improved to show more relevant information to the operators about the clients. As such, the following changes were made to the history screen:

1. The header information was changed to show the client's name, the date of the creation of the client's account, and the number of days that have passed since the client's last call;
2. The header now also includes the number of calls, the average spent per call, the total spent, and the client's top 3 requests for the last 30 days and also for the last year;
3. The list of documents was changed such that the client's last 5 documents are shown, and the very latest one is automatically expanded when the screen is loaded;
4. For each document, the document's repeat button was hidden until it is expanded.

The results of these changes may be seen in Figures 30 and 31.

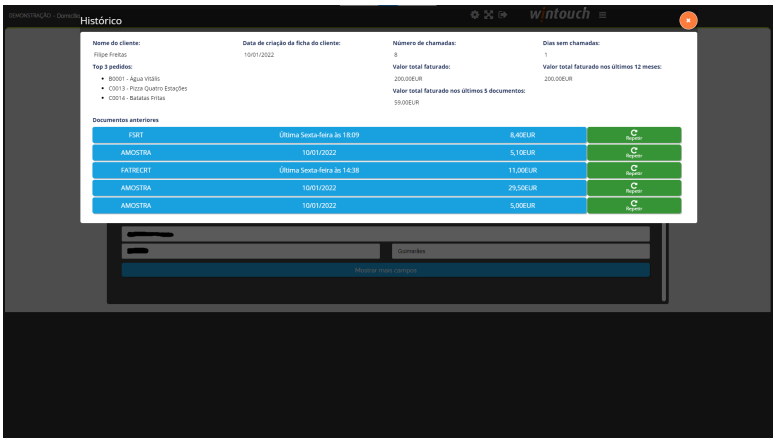


Figure 30: History screen - before changes

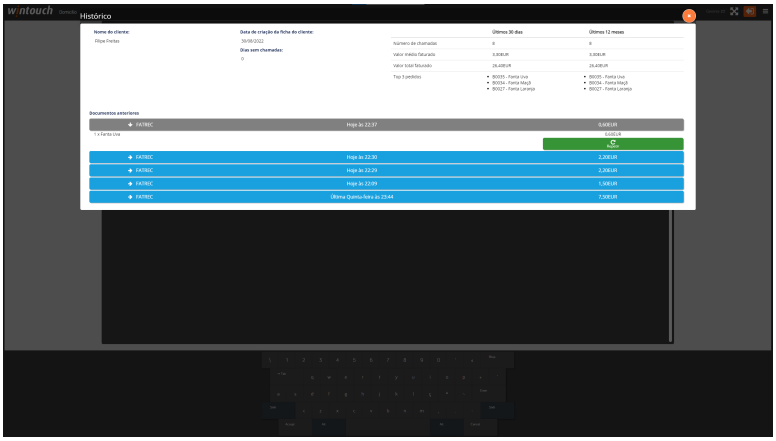


Figure 31: History screen - after changes

6.6 Process of saving new requests

One of the main challenges during the development of this project was the process of saving a new request. Specifically, because requests use an external structure that points to a document that is associated with a request, and considering the fact that our front-end application was not prepared to handle this kind of structure, few options were left to save new requests:

1. For every part of the application that stores documents, the logic behind those parts would have to be changed to handle the new request structure. This would have been a very time consuming process, and it would have also required a lot of testing to make sure that the new structure was handled correctly in all of the parts of the application that use it;

2. Refactor all of the places that store documents, merging all of that separate logic into a central location that can then be changed just once to handle the new structure. This would have been a much better solution, but it would have still required a lot of testing to make sure that the new structure was handled correctly in all of the parts of the application that use it, as well as testing all of the affected parts of the application to make sure that the refactoring didn't break anything;

3. Cleverly intercept the API requests that are eventually invoked before they get to the server, no matter which component or part of the application they come from, and then instead send to the server a different request that contains the new structure. This solution was the adopted solution.

The third solution was the one that was adopted for the following reasons:

- This solution was the fastest, chapest, and least risky to implement;

- This solution requires minimal changes to the parts of the application that save documents. Those solutions are simply to prevent the application from handling the server's response, instead letting the Delivery Service handle it;

- This solution is also, by far, the easiest to test, requiring only tests to the parts of the application that are used in the Delivery module;
- If, in the future, the refactor mentioned above is implemented, this solution can be easily maintained separately (if appropriate), or simply removed, by merging its logic into the refactored code.

The interception of requests makes use of Angular's service injection. Specifically, in Angular, service injection can be made at the Module level, or at the Component level. Since the *DocumentService* class is declared at the component level, it means that, in the Delivery module components, we can specify a different implementation of the *DocumentService* class, which will be used instead of the default one. This is done by declaring a new class that extends the *DocumentService* class, and then overriding the methods that we want to intercept. This new class is then declared in the *providers* array of the component that we want to intercept the requests from. Because child components inherit the providers from their parent components, this means that all of the child components of the component that we declared the new class in will also use the new class. This is the case for the Payments Screen component, which is one of two components that is used to save new requests. The other component is the Request Products component, which, unlike the Payments Screen component, was already under our control. Regardless, both implementations eventually call the *DocumentService*'s *saveDocument*() method, which is the method that sends the API request to the server. Therefore, the new implementation of the *DocumentService* class only overrides this method, keeping the rest of the logic from the original class intact.

However, when the *saveDocument* method is called, it only receives the Document that needs to be saved. However, to save the Delivery Request structure, we must have it accessible from the *saveDocument* method. The way that was achieved that is:

1. In the case of the Payments Screeen, before this screen is opened, the component will save, inside of the Delivery Service, the Delivery Request structure that needs to be saved. This is done by calling the *saveAdditionalDeliveryDetails* method of the Delivery Service. Then, when the *saveDocument* method is called, the Delivery Service will have the Delivery Request structure available, and it will be able to use it to create the new structure that needs to be saved. While this project wasn't designed for a multi-user environment, this solution is also safe to use in a multi-user environment, because requests are correlated to their documents by the document's ID, which is unique for each document. Cleanup is then performed once either the request is saved, or the user closes the Payments Screen;
2. In the case of the Request Products component, the Delivery Request structure is also saved inside of the Delivery Service. However, because we don't have to wait for user action to save the request, after saving the Delivery Request structure in the Delivery Service, we also immediately invoke the *saveDocument* method of the *DocumentService* class. This way, the same logic will be used to save the Delivery Request structure as in the case of the Payments Screen component. Cleanup, in this case, is performed after the request finishes saving.

6.7 Internal API process for initiating deliveries and associated invoice generation

Another challenge in implementing the process of saving requests was the API implementation of the new endpoint for generating invoices for saved requests, when a driver is initiating deliveries, or when the module is configured to immediately generate invoices when a new request is created. Specifically, in the first implementations of this endpoint, it would take about 30 seconds to generate two or three invoices for the requests. That amount of time is completely unacceptable for a multi-tenant server, as well as from the user point of view. Therefore, a solution had to be found for this problem.

However, before attempting to find a solution to this problem, it was attempted to find the cause of the problem. By using profiling tools on the server, it was realized that the problem was caused not by the invoice generation itself (that code was not developed by me, and it was not taking nearly as much time when running outside of the Delivery module), but by the process of getting, from the database, the Delivery Requests and their nested data structures, such that proper changes could be made to these entities, and then saved to the database, as well as returned to the client in the response.

Specifically, the problem was that, for requests with a decent amount of products, for every product that the request contained, the ORM used by our API server would make a separate request to the database to get the product's data. The product's data would then also contain nested structures of its own, which would add even more requests to the database. This would result in an exponentially-growing amount of queries to the database, which would slow down the performance of this endpoint considerably, especially considering the fact that this endpoint is capable of handling multiple requests at once. The ORM does this on purpose, but, unfortunately for our purposes, there is no way to force it to bring all of the nested data in a single query, using JOIN statements on the database.

The solution to this problem turned out to be simple: the ORM only stops using JOIN statements after two levels of nesting. That means that, if nesting inside of the Delivery Request structure more than two levels deep could be avoided, then the problem itself could also be avoided. Turns out doing that was both possible and practical: the nested resources that are required are only required to perform recalculations on the Document, once the delivery starts. However, not every single nested structure is required for the recalculation to work. Because of that, the subset of nested structures required was identified, and only these structures are being loaded now. Because of this, the process of generating invoices for multiple requests is now much faster, and the performance of the endpoint is now acceptable: in the same 30 seconds, the application is now able of saving more than 40 requests, each of them with dozens of lines. This is a huge improvement, and it is therefore now possible to use this endpoint in a multi-tenant environment, especially considering that for the typical amount of requests a driver starts (2-3), this endpoint will take less than a second or two to complete.

It is important to note that invoice generation for documents which are not yet invoices is an extremely expensive process, due to legal requirements that have to be respected and checked at every step of the

process. After these improvements, profiling sessions on the code showed that 90% of the time spent on this endpoint is spent in the invoice generation process, and only 2% is spent in the process of getting the Delivery Request structure from the database. The remaining 8% are spent in the process of preparing the endpoint's response and serializing the updated Delivery Request structure, as well as its nested structures, to JSON. This serialization process was also optimized by using a custom serializer for the Delivery Request structure, which is discussed in Section 6.11.

6.8 Request synchronization between posts using SignalR

The Delivery module is designed to be used in a multi-post environment. This means that, in a single installation of the Delivery module, there can be multiple posts, each of them being able to save requests. However, the requests saved in one post should be visible in the other posts, so that the drivers can see the requests that were saved in other posts, modify them if necessary, as well as begin and end the deliveries of those requests. This is done by using SignalR, which is a Microsoft technology, built into the .NET Framework with client libraries in many other languages, that allows for real-time communication between the client and the server. In our case, the Cloud application already uses SignalR for real-time updates of tables (as in, tables of a restaurant) and other information between entities. As such, it was natural to extend this functionality to the Delivery module, so that the requests saved in one post would be visible in the other posts.

The way SignalR updates work in the Cloud application is pretty simple: there are multiple SignalR hubs. Some of them cover updates to the application's base entities, some cover updates to the application's "datalists" (which are essentially updates to all database tables that can be searched by the end users), and the most important hub we have, for our current purposes, is a hub that notifies other posts of changes to requests made in restaurant tables. When a change is made to a document that is associated to a table, the API detects this, and sends a notification to the SignalR hub, which then sends a notification to all other posts, informing them of the change. The posts then start requesting, individually, from the API, the new requests, and the drivers can see the changes made to the requests in real-time. There is already a lot of client-side code that deals with all of these updates, maintaining an open SignalR connection, among other things. As such, it was not necessary to implement any of this code from scratch, and it was only necessary to add a few lines of code to the existing code, to make it work with the Delivery module.

Specifically, since the SignalR messages that are sent by the API to the posts simply contain the ID of the table that was updated, it was only needed to add a special table ID both on the server and on the client, which would be used to identify the requests that are associated to the Delivery module. This way, the existing SignalR code would be able to detect when a change is made to a Delivery request, and send a notification to the other posts, which would then request the new requests from the API, just like they do for the requests associated to restaurant tables.

This means that the SignalR notifications were fairly simple and straightforward to implement, making use of the extensive infrastructure we already have in place for the Cloud application. However, the approach that the Cloud application uses of requesting every single active Delivery request from the server every time a change is made to a request is not very efficient, and could, if a large enough number of requests is active, cause a lot of unnecessary slowdown in the application.

One fix to this would be to only request, from the server, the specific Delivery request that was updated. However, this would require a lot of changes to the existing code, as it is not currently prepared to only send the ID of a single request, meaning that code would have to change. This would also make the

Delivery module behave differently from the rest of the Cloud application, and as such, this approach wasn't the preffered one. It's worth mentioning that if, in the future, the Cloud application's SignalR architecture is changed to accomodate requesting only the changes from the server, it won't be difficult to implement this in the Delivery module either.

The approach taken was therefore to improve the efficiency of the endpoint that returns all of the active requests. Specifically, the endpoint only returns the minimum amount of information per request that the Delivery module will need to function correctly on the client most of the time. There are, however, times when the Delivery module needs more information than the basic information provided. In those cases, before doing an operation that requires that information, the Delivery module will request the full information for that request from the server. This way, the Delivery module will only request the full information for the requests that it needs, and not for all of the requests that are active. This is a much more efficient approach, and it will allow the Delivery module to scale better with demand, without overloading the server with requests. Another important optimization to this process came in the form of the custom serializer discussed in Section 6.11.

6.9 Accessing the Deliveries Monitor screens without employee login

The Deliveries Monitor screens are inherently designed to not require an employee to be logged in to the Point of Sale application for them to work. Specifically, if someone tries to perform any kind of operation, such as starting a delivery, the application will prompt the user to select which employee is performing the action (or, alternatively, ask them for their PIN). As such, the Deliveries Monitor screens do not require an employee to login beforehand, which is the opposite of the Point of Sale application where the Delivery Monitor screens are integrated. As such, a simple solution to this problem was developed.

Since posts can be configured to grant only excluse access to the Deliveries Monitor screens, if a post is configured in this manner, when prompted for the employee login, the user can just press the "OK" key to access the screens, without having to enter any credentials. This way, the Deliveries Monitor screens can be accessed without having to login to the Point of Sale application. A special button may be added in the future for this action.

6.10 Changing delivery payment methods

Another big challenge in the development of the Delivery module is the changing of the specified payment method of a Delivery request. Specifically, once a driver goes out and actually performs a Delivery, if the client ended up using a different payment method than first indicated, it is necessary to update that information in the system before closing the Delivery. While there is a specific button for that, the

technical implementation of the process of saving this information was challenging, but it boiled down to the following:

1. Use the existing payments screen to allow the driver to select the new payment method;
2. Save the document to the database using the Cloud application's existing methods;
3. Once that saving is complete, use the existing code for closing a Delivery to close this specific Delivery request.

Other methods that were explored were using the existing payment screen to allow the driver to select the new payment method, but instead of using the Cloud application's default code for handling saving the document, instead use the Delivery module's code for saving the document. This would allow the Delivery module to save the document in a way that would be compatible with the existing code for closing a Delivery, and to do it all in just one request to the server. This approach, however, was abandoned once it was realized that the amount of times the payment method would require changes would be very low, and the implementation complexity of this new endpoint would be too much for the small amount of times it would be used. Not only that, even if there's an error changing the method, then the request won't be closed, and the module will instead show an error message.

6.11 Request cancellation & Challenges with serialization of API response objects

This section will discuss the biggest challenge that was faced when implementing the ability to cancel Delivery requests: specifically, when cancelling a request that already has an associated invoice, instead of simply deleting that invoice from the database, there are legal requirements that the invoice may not be deleted. Instead, it must be marked as cancelled, and the cancellation must be logged. This means that the invoice must be saved to the database, but with a different status, and with a cancellation reason. Because of this, the old document would have to be serialized to be sent back to the client, so that its cancellation reason and status could be updated. It was during the implementation of this logic that it became apparent that the existing serialization logic could not efficiently handle this scenario.

Specifically, the previous serialization logic of the API instructed the .NET Framework to use the Newtonsoft.Json library to serialize the response objects provided by the controllers to JSON. This library uses the C reflection API to recursively read all of the properties of an object and serialize its entire structure into a JSON object. However, when serializing objects created by our ORM, this API would have trouble, because it would attempt to access properties that the ORM had not loaded into memory yet. Normally, due to the ORM's lazy loading features, this would not be a problem, as the ORM would load the properties into memory when they were accessed. However, when serializing the object, the properties would be accessed after the ORM's Data Context had already been discarded, and as such, without a

valid context from which to access the database, the ORM would simply throw an exception. Another problem was present if the lazy loading was manually disabled in a data context (as was the case in all of the Delivery module's endpoints, for performance reasons). In this case, the properties to be loaded are specified before any loading is done from the database, and all of the required properties would be brought along with the objects, as they were loaded from the database. In this case, the serialization logic does not fail; instead, it simply puts those properties into the JSON respose object, but with their values being null. This is completely undesirable, because of its interaction with our client-side application, which uses an Identity Map design pattern to hold these entities in memory. If the client-side application receives a response object with a null value for a property that it already has in memory, it will overwrite the value in memory with the null value, effectively deleting the value from memory. This is a problem, because the client-side application will then have to request that value from the server again, which will cause a lot of unnecessary requests to the server, and will also cause the client-side application to be out of sync with the server. In some cases, it might not request the value again, and instead, it will just use the null value, which will cause the application to behave incorrectly, or possibly outright crash.

The Cloud application's solution to this problem was to serialize the response objects manually, then de-serialize them, and the resulting object would then be returned to the .NET Framework to be, once again, serialized, this time, to be included in the HTTP Response. This solution proved to be extremely inefficient and slow, because, if lazy loading was enabled, it would simply query the database for every property that wasn't yet loaded into memory. This would work because, since the serialization was being done before the disposal of the data context, the ORM would be able to access the database without any issues. However, this would cause a lot of unnecessary database queries, which would slow down the response time of the API. If lazy loading was disabled, then the serialization would simply send null values to the client, which would cause the client-side application to behave incorrectly, as described above.

Because of the problems described, it was instead decided to implement a custom serialization routine using the Newtonsoft library's APIs. This routine doesn't actually perform the serialization (the library still handles the serialization itself), but instead, it is used to filter out the properties that should not be serialized. This routine is implemented as a custom ContractResolver, which is a basic class part of the Newtonsoft library. A ContractResolver contains many methods, but the one that was necessary to be overriden was the CreateProperty method, which is a method that, given a Member of a class that is being serialized as well as some metadata about it, should return a JsonProperty object that specifies information about the member being serialized, such as the name of the property, its value, whether it should be serialized, and so on. This method is called for every property of the object being serialized, and the returned JsonProperty object is then used by the Newtonsoft library to serialize the property.

In the overriding of this method, it is first checked that the property being serialized represents the value of a foreign key relationship, as used by the ORM (these properties are marked by the ORM with a special attribute, making them easy to identify). If they are, then the returned JsonProperty's ShouldSerialize method is overriden. This ShouldSerialize method is called during the actual serialization of an object, and it is provided with the instance of the object being serialized. The first thing this method does

is check if the property's value can be normally accessed, without an exception being thrown. If it can, then the property is serialized normally. If it can't, then the method checks for one of two conditions: if the property represents a list of objects that have this object as a Foreign Key (for instance, a list of Payments on a Document would be a property of this type), or if represents a one to one relationship. Regardless of the case, it uses the ORM's internal mechanisms (accessed through Reflection) to check if the property is loaded into memory (or, in the case of a list, if all of its values are loaded into memory). If yes, then it serializes the property; otherwise, it does not serialize the property.

This method is much more efficient than the previous method, because it does not query the database for every property that is not loaded into memory, thereby not sending it to the client, and it also does not send null values to the client, which would cause the client-side application to behave incorrectly. This is because the client-side application will not override locally-loaded properties that are not present in the response object, and instead, it will simply ignore them, continuing to use the locally-loaded objects. This method also avoids having to serialize and de-serialize the response objects twice, which was the case with the previous method, making it much faster, even with the slightly increased Reflection (mostly because this extra Reflection is only used for properties that have Foreign Key relationships).

As a final note, it must be stated that this custom serializer is only used on the endpoints specifically marked to use it (such as all of the Delivery module's endpoints), thereby preventing any other parts of the application from breaking due to these changes.

After all of these changes, it was noticed, in profiling, that the endpoints for beginning and closing deliveries only spent about 10% of their time in the serialization logic, and the rest of the time was spent actually doing the useful work of querying the database and updating the objects that exist there with the necessary changes. This is a huge improvement over the previous state of these endpoints, which spent over twice that amount of time in the serialization logic (which makes sense, considering they were serializing the entire object graph, and potentially making extra, unnecessary queries to the database while doing so, instead of just serializing the necessary properties).

These changes were then applied to the other endpoints in the Delivery module, and the same improvements were observed. Not only that, as the Delivery module gets merged into the main development branches, its possible this serializer will become the standard for other endpoints that run into the same issues as well.

6.12 Request editing

One of the last challenges of the Delivery module's main implementation was the process for editing an active request, or more specifically, for saving an updated version of the request in the database.

The initially planned process for saving an updated version of the request would be to detect if the request's associated document was already an invoice, and if yes, cancel the old document (as is legally required), and create a new one (identical to the old one, but with the changes made by the user), then

update the foreign key relationship to point to the new document.

This, however, proved problematic, especially with the incredibly high number of different cases that would have to be dealt with. Not only that, due to the way transactions are internally implemented inside of the API, it meant that the API's document saving logic was trying to delete a document that still had a foreign key pointing to it, which would cause an exception to be thrown, and the entire transaction to be rolled back. It was also tried working around this by manually deleting the foreign key relationship BEFORE the API's internal logic kicked in to delete the document, but unfortunately, it would internally reorder things to delete the document first, and then delete the foreign key relationship, which would cause the same exception to be thrown.

Another attempt at fixing this was simply to allow null values for the foreign key relationship, have the API manually prohibit users from submitting null values, and, after the document was deleted, update the foreign key relationship to point to the new document. This would work, but it would cause the database changes saving logic to, sometimes, not properly update the foreign key value's in the database, and since there was nothing preventing it from having a null value there, it would accept it without any problems. This would cause the API to return an incorrect value for the foreign key relationship, which would cause the client-side application to behave extremely incorrectly.

As such, it was decided to instead always delete the request and its associated document. The internal reordering, in this case, isn't an issue, because the deletes are executed by the API in the order that they are given, unlike the updates, which would always get placed after the deletion of the document (which is what caused the issue in the first place). After that, a new identical request would be created (but with the updated foreign key), and the new document would be created, also with the correct updates already applied. This also simplifies the number of individual cases that have to be handled down to simply just one.

6.13 Delivery Zones

The initial approach taken towards this goal was decided upon months before its actual implementation, and it was decided that the best way to implement this would be to use the Google Maps API, and to use the Google Maps API's geocoding functionality to convert addresses into latitude and longitude coordinates, and then figure out if the coordinates are inside of a given zone. Zones would be pre-configured by a restaurant, and they would be drawn on a map. Those drawings would result in one (or, more likely, multiple) polygons being created, and those polygons would be used to determine if a given address is inside of the zone or not, upon the user inserting a client's address.

As it turns out, however, in the time between conception and implementation, Google Maps has changed their pricing model, and now, the geocoding API is no longer free, and instead, it costs $5 per 1000 requests[1], at the time of writing. Even though the original solution was already planning on caching the geocoding results for some period of time, this still proved to be a problem due to cost.

As such, other solutions had to be considered. The first obvious solutions considered were to look for alternative APIs that could have the same accuracy as Google Maps, and that could keep the costs down. Unfortunately, none of the alternative APIs explored were able to meet both of these requirements.

There was, however, another alternative: instead of using an external API for geocoding, we could instead use the freely-available OpenStreetMap data files to do the geocoding ourselves. This simply requires pre-loading our production databases with the relevant OpenStreetMap information, which means some extra work was required to read the OpenStreetMap files, extract the relevant data, and generate the necessary SQL queries to insert it into our production databases. This is a one-time cost, however, and it would allow us to avoid the $5 per 1000 requests cost, and instead, we would only have to pay for the cost of the database storage, which is much cheaper. Updates to this database could also be integrated into the API, so that the API would automatically update the database with the latest OpenStreetMap information periodically, and the API would also automatically update the database with any changes made by the restaurant's administrators.

Even this approach, however, proved problematic: the OpenStreetMap data files are very large, and they are also very complex, and it would take a lot of time to read them, extract the relevant data, and generate the necessary SQL queries to insert it into our production databases. Not only that, the geocoding process itself is also very difficult to implement, because, for example, it would require implementing full text searching capabilities from scatch into the Cloud API. While there are libraries that can definitely help with these problems, it would still take a lot of time to implement, and it would also require a lot of testing to ensure that the geocoding process is accurate enough to be used in production. This would be necessary because the company requested that no new servers are used, so as to not add any extra cost by the Delivery module.

This last condition also precludes the use of Nominatim[2], which is a free and open source tool for

[1] https://mapsplatform.google.com/pricing/
[2] https://nominatim.org/

geocoding using OpenStreetMap data. This tool is very easy to use, and it would be able to do the geocoding for us, but it would require the use of a new server to host it, as well as a new database server to store the specific data required, which would violate the condition set out by the company.

As such, it was have decided to opt for an alternative: instead of using the OpenStreetMap files, or even attempting to do any geocoding at all, we would opt for a simpler approach, already present in the Desktop application (albeit with some minor efficiency improvements). This approach is to use a pre-existing database of all of the addresses and postal codes existing in Portugal (provided by the Portuguese Postal Service), and, when creating a zone, a user is able to filter those addresses by some criteria, and then, the addresses that match the criteria are added to the zone.

Unlike in the Desktop application, however, the addresses do not need to be stored in the restaurant's internal database; instead, there can exist a global database, shared by all of the restaurants, and, when any restaurant needs to match an address against a zone, it simply needs to query the global database to figure out if the address is valid, and, if it is, it can then query the restaurant's internal database to figure out if the address is inside of a zone or not, by checking whether the given address's ID is present in the zone's list of addresses.

Chapter 7

Final working product

In this chapter, it will be provided an overview of the product in its current state, which is a mostly finished state (pending fixes to bugs that might appear), illustrated with screenshots from the application collected during its usage.

7.1 Registration of New Requests

In the point of sale application, an employee with permissions may enter the new registration request screen. The employee will see a screen showing a text box, illustrated in Figure 32 where the employee may search for a client by name, code, phone number, or other properties.

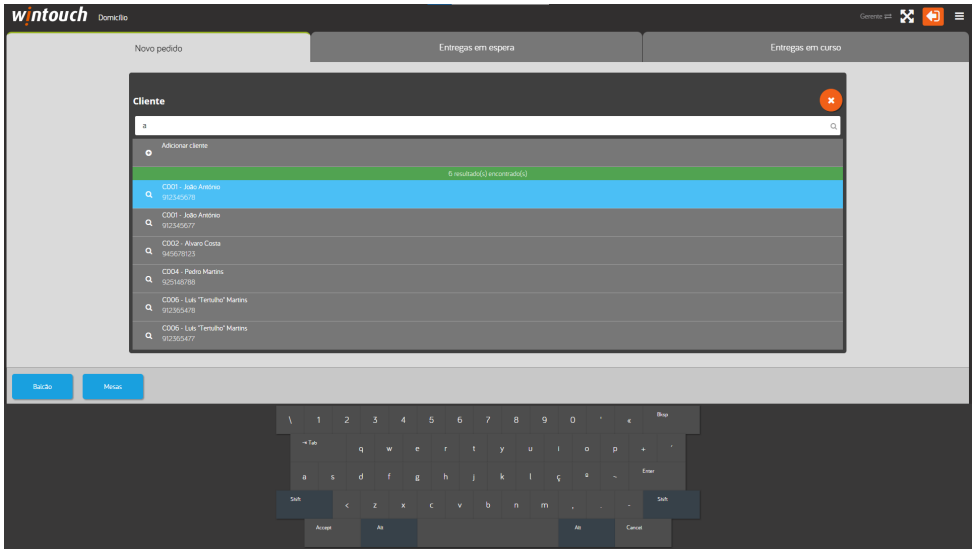


Figure 32: Request registration - Client Search

From that screen, the employee may choose a client to register a new request for, or may choose to register a new client. Regardless of which option they choose, the employee will be presented with a screen where they may enter the details of the request, as illustrated in Figure 33.

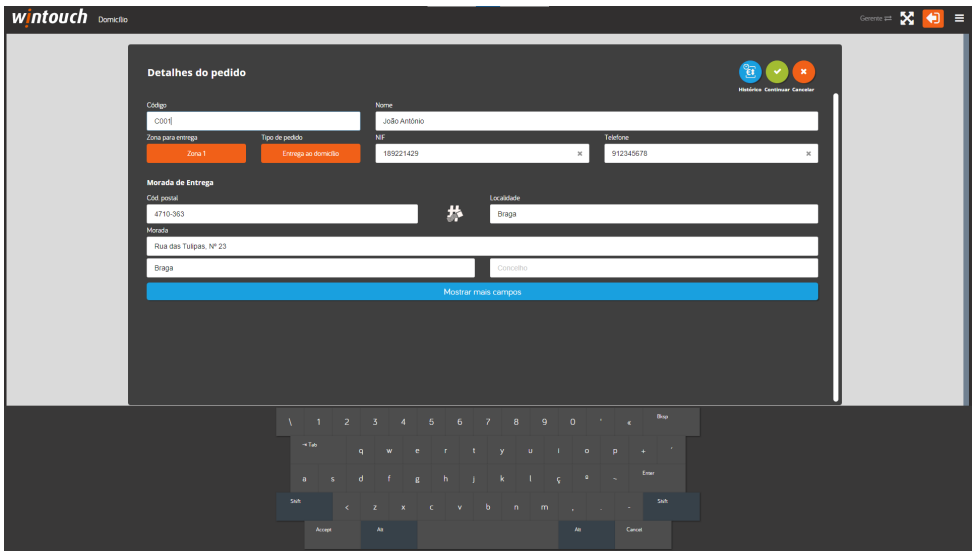


Figure 33: Request registration - Insert Request Details

From that screen, the employee may then choose to visualize the client's history, as illustrated in Figure 34.

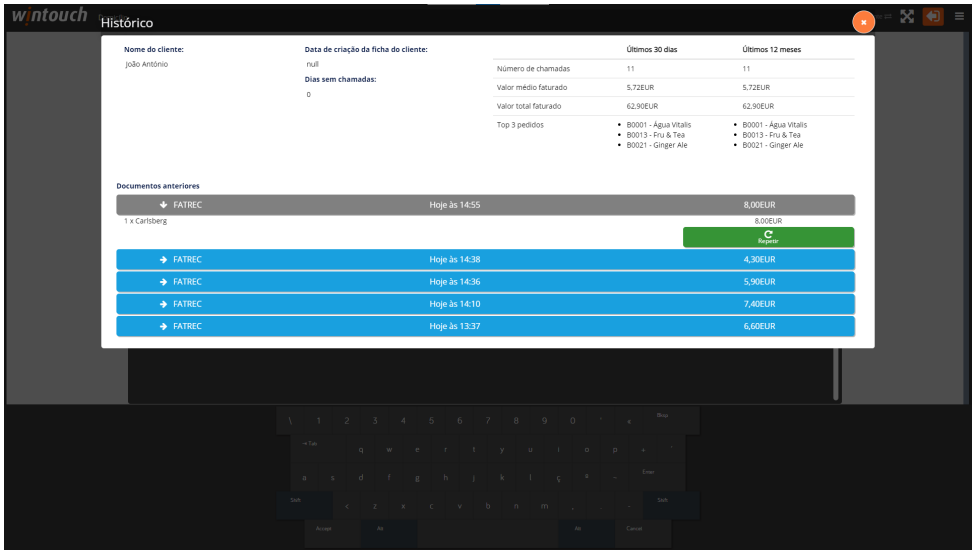


Figure 34: Request registration - Client History

The employee may then choose to proceed to the next page (Figure 35), where the employee is able to insert the various items that the client has requested.

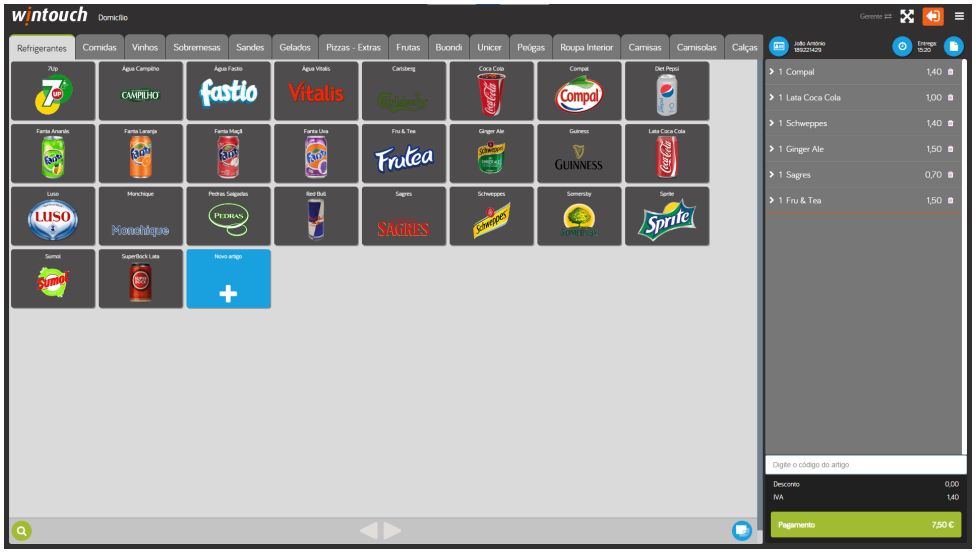


Figure 35: Request registration - Insert Request Items

The employee will then introduce information on how the client intends to pay for the request. The system will then save and submit that information to the server. This is illustrated in Figure 36.

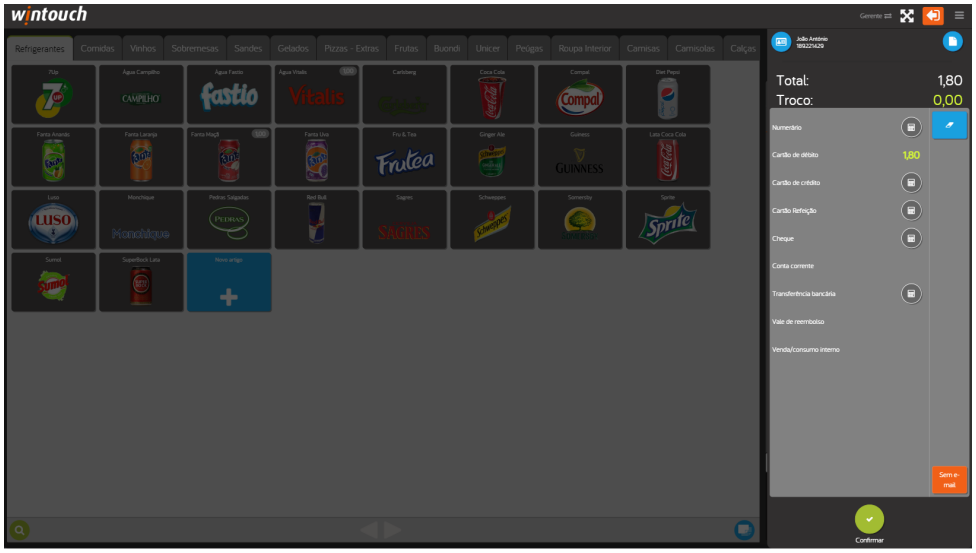


Figure 36: Request registration - Payment

If the employee selects a client that already has an active request, however, they will not be allowed to insert a new request. Instead, they will be shown a list of options to operate on the existing request, as illustrated in Figure 37.

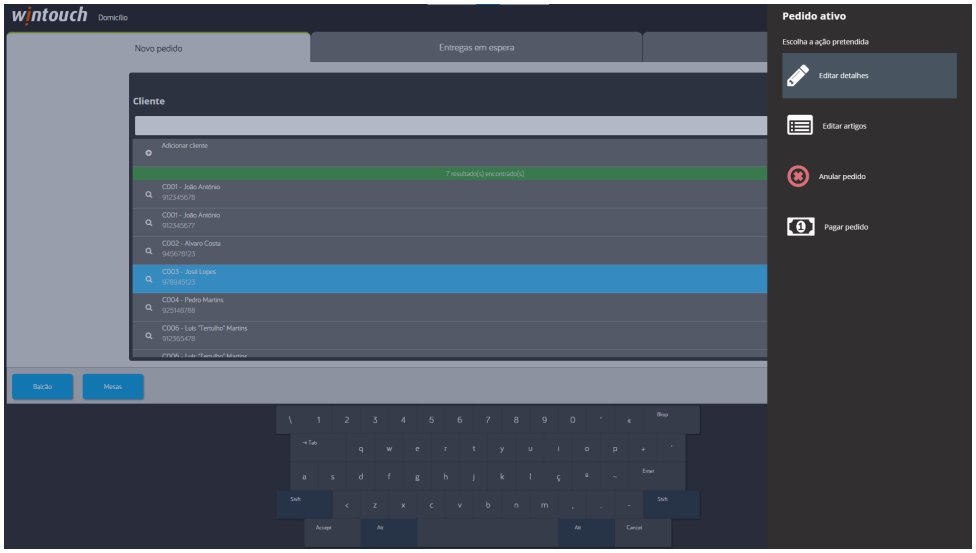


Figure 37: Request registration - Existing Request

Upon selecting the desired action, the employee will then be presented with the required screen for performing the given action, or, if the action requires no special screen, the employee will simply see a success or error message (depending on the result of the action).

7.2 Requests waiting for delivery

In another tab, selectable from the top or accessible through other posts, the employee may choose to view the active requests that are not yet in delivery. The employee will then be presented with a list of requests, as illustrated in Figure 38.

Figure 38: Requests waiting for delivery - List

Figure 38 shows the list of requests that are waiting for delivery, as well as the list of zones that the requests are in. The employee may start to select requests at will, by clicking on them, or the employee

may select all requests in a zone by clicking on the zone's name. The employee may then choose to start the delivery of the selected requests, by pressing the buttom in the bottom right corner of the screen, as illustrated in Figure 39.

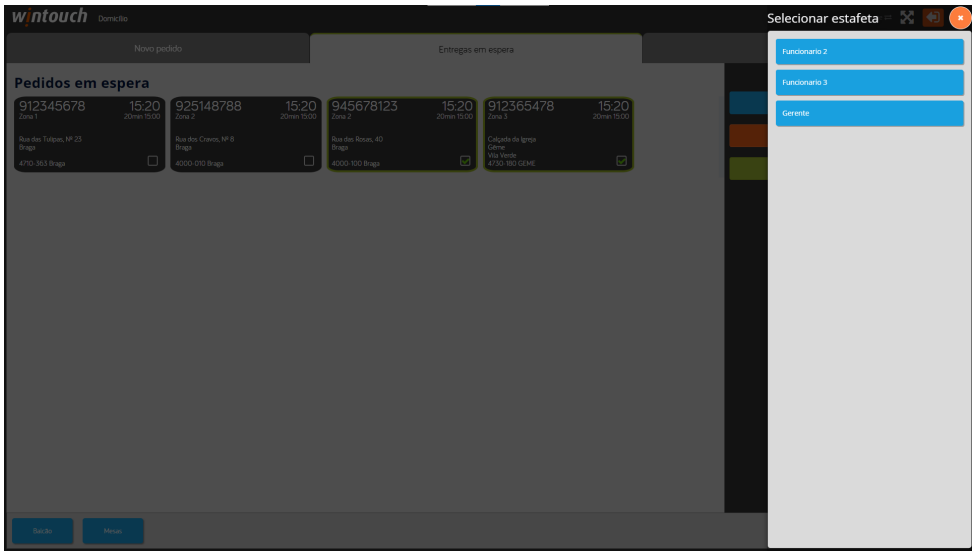


Figure 39: Requests waiting for delivery - Selecting requests - List

Because the screen in Figure 39 was designed to be used without an employee logged in, when the employee attempts to perform the "Begin Deliveries" action, the employe will be asked for identification, by selecting his/her name from the list of available employees.

However, if the restaurant has configured the Delivery module to require PINs for employee actions, the employee will instead be asked to enter his/her PIN before proceeding, as illustrated in Figure 40.

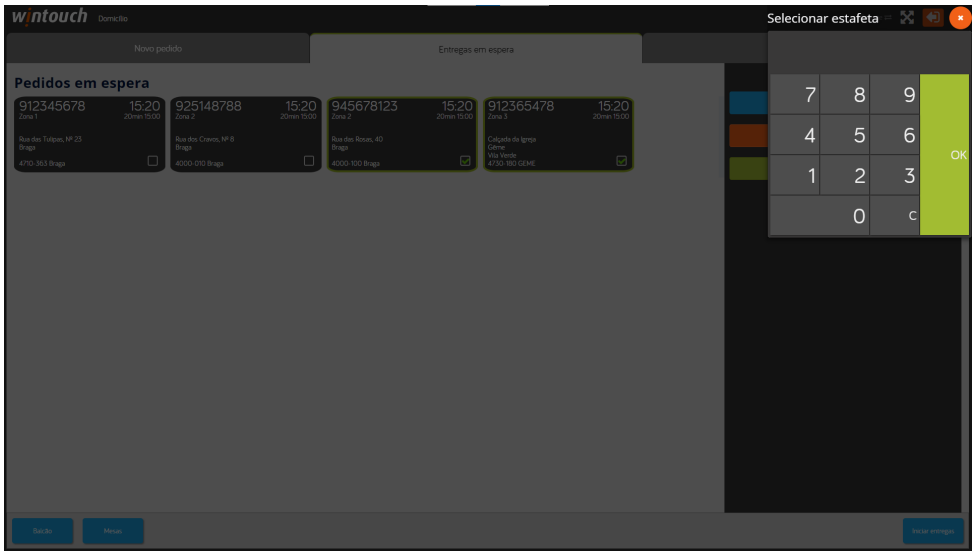


Figure 40: Requests waiting for delivery - Selecting requests - PIN

After the employee identifies him or herself, the requested deliveries will then be marked in the system as started, and the employee will be returned to the list of pending requests.

7.3 Requests in delivery

In a different tab, also accessible from the top, when inside of the Delivery module, the employee may choose to view the active requests that are in delivery. In the normal mode (where employees are not required to insert their PINs), a list of employees will be shown. Employees will then be able to perform actions on the requests that they are currently delivering, such as marking them as delivered, cancelling them, returning them, or changing the payment method of a request before marking it delivered, as illustrated in Figure 41.

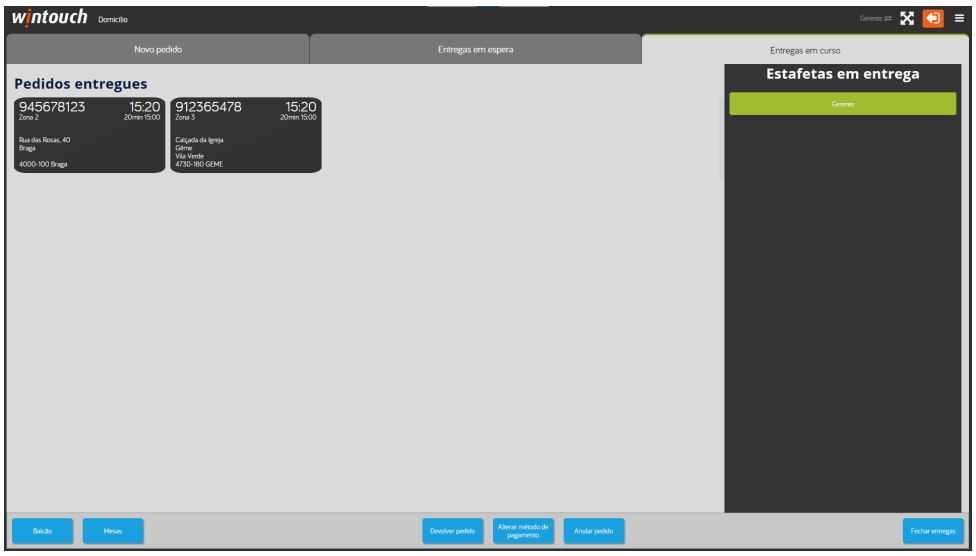


Figure 41: Requests in delivery - List

However, the list of employees will not be shown if the restaurant has configured the Delivery module to require PINs for employee actions. In this case, the employee will be asked to enter his PIN before proceeding, and, once they do, they will only be shown their own requests, as illustrated in Figure 42.

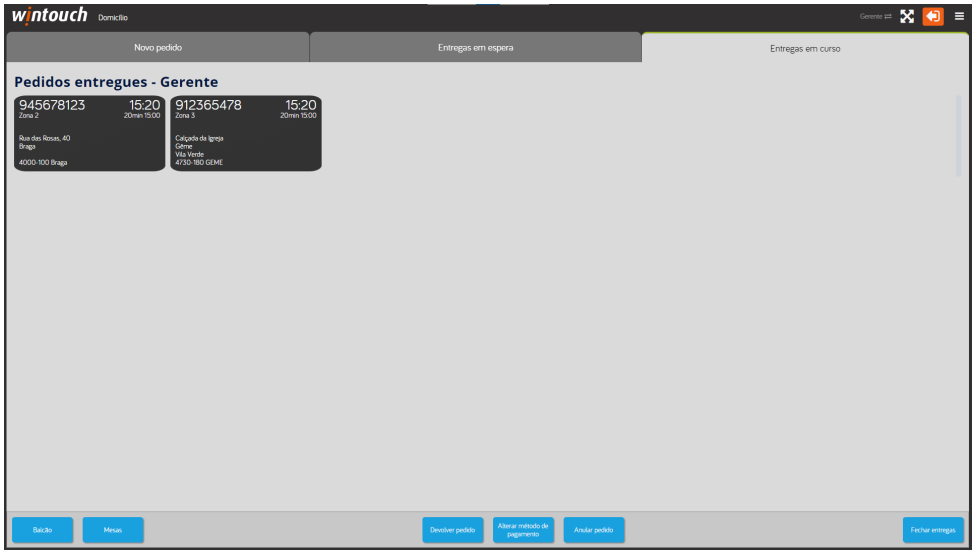


Figure 42: Requests in delivery - PIN

After identifying themselves, and without having to do it again until exiting the screen in Figure 42, the employee will be able to perform any action he/she desires on the requests that he/she is currently delivering. The available actions are the same as above.

There are other screens that were not shown here, such as the screen for creating a new zone, or for creating a new Delivery Order Type. However, these screens were not completed enough for them to be shown here, and as such, they will not be shown.

Chapter 8

Conclusion

At the start of this project, it was proposed to attempt to tackle the problem of creating a Delivery module in a new and innovative way, that could improve upon the already existing methods in the market, and that could deliver a better, more innovative product for Wintouch's customers, at a lower cost.

As such, after accepting this challenge, work started on reviewing the existing literature on the subject, and on the existing products in the market, in order to identify the main problems that the existing products had, and to identify the main features that the new product would need to have in order to be successful. A deep dive into the existing solution created by Wintouch in their Desktop product was also made, in order to identify the main problems that the existing solution had, and also to help with identifying the main features that the new solution would need to have in order to be successful.

It was then proposed a new solution for the Delivery module, that would be based on the existing solution, but that would be improved upon in order to address the main problems that the existing solution had. After many revisions and iterations, it was proposed the solution that was explained throughout this document, and that was implemented in the final working product.

During the development of the new solution, many problems occurred, and while some were predicted, many were not. However, after research and after iterations, all of them were overcome in a way that satisfied the company. The main problems solved were also presented in the previous chapters, as well as explanations of the process followed in order to solve them, and overviews of the solutions implemented.

Finally, in the last chapter, it was presented the final working product, and it was also presented the main features that it has.

Therefore, it is believed that all of the main objectives that were given were completed, and it also believed that a new and innovative solution for the Delivery module was created, that is better than the existing solutions in the market, and that is also better than the existing solution in the Desktop product.

However, a secondary objective was not completed: the Delivery Zone mapping using Google Maps, for the reasons also explained above. However, and also according to feedback received from the company, this is not a critical feature, and therefore, it is not a problem that it was not implemented. The main features that were implemented are more than enough to make the product successful, and therefore, it is believed that the project was a success. This is especially true considering that a substantial portion of the time at the company was spent outside of this project, and that the project was still completed in a timely manner.

In terms of future work, it is believed that with more time, the Delivery Zone mapping feature could eventually be integrated into the product at a cost that would be acceptable to the company. It is also believed that other features that were explored during the State of the Art research could also eventually be implemented, such as Kitchen Monitors (screens, present in a restaurant kitchen, that automatically update with the orders that are being prepared, and that also allow the kitchen staff to mark the orders as being prepared, and to mark the orders as being ready to be delivered), and also the ability to send the orders to the kitchen staff via a mobile app, instead of via a printer. Other interesting features to implement would be social media interactions (a Social media bot that would allow customers to place orders via Social Media, a very popular feature in other parts of the world, such as China and Brazil), reservation centers (or, in other words, call centers that handle all calls for a restaurant chain and send the orders to the restaurant belonging to a chain closest to the customer), and possibly also a food waste app, that would allow customers to order food that would otherwise be thrown away, at a lower price, in order to reduce food waste.

Bibliography

[1] A. M. Florio, D. Feillet, and R. F. Hartl. "The delivery problem: optimizing hit rates in e-commerce deliveries". In: *Transportation Research Part B: Methodological* 117 (2018-11), pp. 455–472. issn: 01912615. doi: 10.1016/j.trb.2018.09.011 (cit. on p. 12).

[2] Z. He et al. "Evolutionary food quality and location strategies for restaurants in competitive online-to-offline food ordering and delivery markets: An agent-based approach". In: *International Journal of Production Economics* 215.April 2018 (2019), pp. 61–72. issn: 09255273. doi: 10.1016/j.ijpe.2018.05.008 (cit. on p. 1).

[3] K. Kollnig et al. "Goodbye Tracking? Impact of iOS App Tracking Transparency and Privacy Labels". In: (2022-04). doi: 10.1145/3531146.3533116. url: http://arxiv.org/abs/2204.03556%20http://dx.doi.org/10.1145/3531146.3533116 (cit. on p. 13).

[4] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. url: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[5] C. Ribeiro. "Technology at the table: an overview of Food Delivery Apps". Master's Thesis. Universidade Católica Portuguesa, 2018, p. 79 (cit. on p. 1).

[6] A. Seghezzi, M. Winkenbach, and R. Mangiaracina. "On-demand food delivery: a systematic literature review". In: *International Journal of Logistics Management* (2021). issn: 17586550. doi: 10.1108/IJLM-03-2020-0150 (cit. on p. 1).