FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Mobile application and tool for the study and prevention of hypoglycaemia after bariatric surgery

João Filipe Afonso Martins

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: António Miguel Pimenta Monteiro

July 31, 2022

Resumo

A obesidade foi reconhecida como a pandemia de doenças do século XX, com mais de metade da população em Portugal a ser afectada por excesso de peso ou obesidade. O tratamento mais eficiente para pacientes diagnosticados com obesidade mórbida é a cirurgia bariátrica. Com este procedimento cirúrgico, é possível obter uma perda de peso a longo prazo e a remissão ou atenuação de doenças relacionadas com a obesidade. Este tratamento tem muitos benefícios que são geralmente mais consideráveis do que os riscos associados, que também podem existir, um dos quais é a hipoglicémia pós-bariátrica. Sempre que houver suspeita de que um paciente sofre desta complicação, ele precisa de um acompanhamento diferente, sendo necessário controlar a sua glicemia ao longo do dia, bem como saber o que comeu e quais os sintomas experienciados durante o dia. O objectivo é obter uma melhor confirmação do diagnóstico, avaliar a gravidade da condição e monitorizar a resposta do paciente aos vários tratamentos. Para que tal seja possível, foi fornecido ao paciente um sistema de monitorização da glicose, o Freestyle Libre 2, que permite ao utilizador deste sistema ter um maior controlo da sua glicemia sem ter de recorrer à sua verificação com sangue, o que o torna um método não invasivo. Para além deste sistema, o paciente é solicitado a manter um registo detalhado das refeições que come para estimar a energia total consumida diariamente e registar todo o tipo de sintomas por ele experimentados ao longo do dia.

Esta dissertação visa preencher a lacuna no mercado relativamente ao tratamento da hipoglicemia após a cirurgia bariátrica, desenvolvendo uma aplicação móvel que permite ao utilizador registar as suas refeições diárias, bem como os sintomas observados ao longo do dia, e fornecer ao utilizador a informação da glicose sanguínea. A aplicação foi desenvolvida utilizando Flutter e pacotes pré-fabricados desenvolvidos pela comunidade. A arquitectura Redux foi utilizada para a arquitectura da aplicação, que tem um pacote do Flutter para uma melhor utilização no desenvolvimento. Após a produção da aplicação, houve ainda a necessidade de tratar os dados introduzidos pelo utilizador para que pudessem ser visualizados pelo profissional de saúde, surgindo a necessidade de criar um servidor para tratar estes dados. Este servidor foi desenvolvido em Nodejs, que, tal como Flutter, tem pacotes que podem ser utilizados para facilitar a implementação.

Por fim, foram realizados alguns testes para compreender se a configuração desenvolvida cumpre o pretendido, bem como a inserção de refeições e sintomas para garantir o cumprimento das regras de cada ecrã.

Abstract

Obesity was recognised as the 20th century disease pandemic, with more than half the population in Portugal being affected by overweight or obesity. The most efficient treatment for patients diagnosed with morbid obesity is bariatric surgery. With this surgical procedure, it is possible to achieve long-term weight loss and remission or attenuation of obesity-related diseases. This treatment has many benefits that are generally more considerable than the associated risks, which may also exist, one of which is post-bariatric hypoglycaemia. Whenever there is suspicion that a patient suffers from this complication, he needs a different follow-up, being necessary to monitor his blood glucose throughout the day, as well as know what he ate and what symptoms were experienced during the day. The goal is to get a better confirmation of the diagnosis, assess the severity of the condition, and monitor the patient's response to the various treatments. For this to be possible, the patient was provided with a glucose monitoring system, the Freestyle Libre 2, which allows the user of this system to have greater control of their blood glucose without having to resort to checking it with blood which makes it a non-invasive method. In addition to this system, the patient is asked to keep a detailed record of the meals he eats to estimate the total energy consumed daily and record all kinds of symptoms experienced by him throughout the day.

This dissertation aims to fill the gap in the market regarding the treatment of hypoglycemia after bariatric surgery, developing a mobile application that allows the user to record their daily meals, as well as the symptoms experienced throughout the day, and provide the user with the information of blood glucose. The application was developed using Flutter and pre-made packages developed by the community. The Redux architecture was used for the application's architecture, which has a Flutter package for better use in the development. After the production of the application, there was still the need to treat the data entered by the user so that they could be visualised by the health professional, arising the need to create a server to treat these data. This server was developed in Nodejs, which, like Flutter, has packages that can be used to facilitate the implementation.

At last, some tests were performed to understand if the developed configuration meets what was intended, as well as the insertion of meals and symptoms to ensure compliance with each screen's rules.

Agradecimentos

Em primeiro lugar gostaria de agradecer aos meus orientadores, o professor António Monteiro e a professora Mariana Monteiro por toda a disponibilidade e ajuda na realização deste projeto. Quero ainda agradecer ao Bruno e à Carolina pela sua grande ajuda, paciência e disponibilidade ao longo deste percurso.

Como não podia deixar de mencionar, agradeço especialmente aos meus pais por todo o apoio incondicional que me transmitiram, nunca duvidando das minhas capacidades e particularmente por me providenciarem esta oportunidade. Sem eles todo este percurso teria sido mais difícil e desmotivador. À restante família que, direta ou indiretamente, fizeram parte desta caminhada o meu muito obrigada. A força que sempre manifestaram fez, certamente, a diferença.

À Márcia Rafaela por toda a paciência que tem comigo, por toda a ajuda e apoio que demonstrou no decorrer deste percurso que, sem ela, não teria sido o mesmo. Por todo o amor, carinho e atenção um muito obrigado a esta pessoa incrível que sei que terei sempre a meu lado.

Um agradecimento especial ao meu primo Fernando por toda a ajuda, disponibilidade e ensinamento prestados durante a minha etapa académica.

Aos meus amigos, aos sempre 17, que estiveram sempre presentes e mostrarem sempre o seu apoio incondicional.

A todos aqueles que não constam nos mencionados acima, mas que ao longo deste percurso cruzaram o meu caminho e que de certa forma contribuíram para que esta jornada fosse melhor, o meu muito obrigado.

João Filipe Afonso Martins

"Na vida, não existem soluções. Existem forças em marcha:
é preciso criá-las e, então, a elas seguem-se as soluções"

Antoine de Saint-Exupéry

Contents

List of Figures

List of Tables

Acronyms and Abbreviations

$ADRR_{FGM}GT$	Average Daily Risk Range (adjusted)
CNN	Convolutional Neural Networks
CONGAn	Continuous Overlapping Net Glycemic Action over an n-hour period
FSL	Freestyle Libre
FGM	Flash Glucose Monitoring
$HBGI_{FGM}GT$	High Blood Glucose Index (adjusted)
IFG	Interstitial Fluid Glucose
IQR	Interquartile Range
$LBGI_{FGM}GT$	Low Blood Glucose Index (adjusted)
MAG change	Mean Absolute Glucose change
MODD	Mean of Daily Differences
PBH	Post-Bariatric Hypoglycaemia

*/

Chapter 1

Introduction

The report presented here intends to give a description for the Dissertation proposal: “Mobile application and tool for the study and prevention of hypoglycaemia after bariatric surgery”.

1.1 Context

This dissertation proposal results from the unmet need of having an efficient and user-friendly tool to record daily food intake and blood sugar levels simultaneously, to enable crossing the data in order to analyse the food drivers that lead to low blood sugar levels –hypoglycaemia - in patients who have been subjected to bariatric surgery, i.e. surgery for obesity treatment. To carry out these diagnostic medical assessments, the standard practice consists in asking the patients to conduct a manual record of daily food intake and symptoms for 14 consecutive days. These recorded data is then manually crossed with the interstitial blood glucose values measured every 15 minutes for the same time period using the Freestyle Libre system. The current approach, despite being highly informative and feasible, is cumbersome and highly time consuming. This limits the use of this approach beyond research scenarios and widespread implementation in routine clinical practice.

1.2 Motivation

Taking into account the great need for a tool that automates the process described above as much as possible, the subject of this dissertation arose. With this, it is intended to provide patients with a mobile application in order to enable the recording of meals and symptoms throughout the days. In this way it will be possible, together with the blood glucose values monitoring, to estimate the impact of meal composition on blood sugar and thus predict and prevent hypoglycaemia. Obesity was recognized as the 20th century disease pandemic, which persists in current century despite the resurgence an infectious disease as a major threat to public health[1]. Moreover, obesity is also a major risk factor for high morbidity and death due to COVID-19. Though it is crucial to pursue the efforts to address obesity treatment as a major scourge of the current century. According to the OCDE, individuals with obesity can suffer a 2.7-year reduction in average life expectancy and, in

addition, more than half of adult Portuguese population present overweight or obesity. Bariatric surgery is currently the most effective treatment for patients diagnosed with morbid obesity since this surgical procedure enables to achieve long-term weight loss and remission or attenuation of obesity-related diseases. Although the benefits of surgical intervention largely outweigh the risks, it is not devoid of complications, such as post-bariatric hypoglycaemia. Whenever there is a clinical suspicion of this medical condition, a glucose monitoring system is installed, and the subjects are asked to record a detailed handwritten description of food consumed, the time at which they ate it, together with any symptoms experienced after meal. This record should contain detailed information in order to be able to estimate the total energy consumed daily. Cross matching the information with blood glucose levels, will the enable to confirm the diagnosis, assess the severity of the condition, and monitor the response to treatment interventions.

1.3 Objectives

This project aims to develop an application capable of linking the various processes required for a better assessment of the user. Firstly, it will be essential to design an information system capable of storing not only the blood glucose values from the Freestyle Libre system, but also each user's meal records. Achieving what is intended will be an asset when it comes to the management of patients who have undergone bariatric surgery, allowing to fine tune disease diagnosis, severity assessment, and monitoring treatment interventions not only by the health professional but also by the empowered patient, towards improved clinical and patient related outcomes.

Chapter 2

Literature Review

The main objective of this section is to review the fundamental theoretical material needed to gain a better understanding of the medical condition so called post-bariatric hypoglycaemia. This includes topics such as, Glycemic Variability, Interpretation of Continuous Glucose Monitoring Data, Quality of Glycemic Control etc. This section also aims to conduct state-of-the-art review. It is important to understand previously existing solutions that were conceived with the aim of solving the same problem. Due to the fact that this is a medical issue it was necessary to carry out further research on the subject. This means that the cited studies were very useful, being necessary to take knowledge from all of them in order to build a solution.

2.1 Background

To date, hypoglycaemia occurring in individuals after being submitted to bariatric surgery has been addressed by several studies. These have been used as a basis for the initial development and research on this topic. As already mentioned, as obesity prevalence continues to rise, so dues the number of candidates for bariatric surgery increase, since this is the most effective treatment for severe obesity [2]. Despite the safety and benefits of bariatric surgery it is not free of risks of early and late complications, such as post-bariatric hypoglycaemia (PBH), which notwithstanding being rare is becoming increasingly recognized as the number of patients that underwent the procedure also rises. Patients who were under surveillance at a multidisciplinary bariatric surgical centre that presented with clinical features suggestive of hypoglycaemia after undergoing bariatric surgery were enrolled in a study comprising glucose monitoring and food and symptoms diary record. For glucose monitoring, users were equipped with the flash glucose monitoring system Freestyle Libre, according to the manufacturer's instructions. The Freestyle Libre system is composed of a sensor housed in the subcutaneous tissue, under the superficial layers of the skin, which automatically measures and continuously stores interstitial glucose levels readings throughout the day and night. The sensor collects and stores the information up to a maximum of 8 hours before being passed to the reader. This minimally-invasive system allows the user self-assess glucose levels without using the traditional method to obtain this value, by using a blood glucose meter that analyses a small

drop of blood that is taken by pricking the fingertip. After installing the glucose reading system, the subjects were asked to record a food diary, including the time of the meals, together with any symptoms experienced afterwards, if occurred. This record should contain detailed information of all the food eaten in order to enable to estimate the total energy consumed. The data obtained from the meal records were then superimposed on the graphs recorded by the sensor, thus allowing a more precise and comprehensive analysis of the condition. As this is a time-consuming and non-automated process, thus the need arose to develop a tool capable of linking the readings of the glucose sensor information to the meal diary records required for a better understanding of PBH, so that the diagnosis could be confirmed, the severity of the condition determined and the treatment could be better monitored to improve clinical outcomes.

2.2 Problem characterization

Bariatric surgery is worldwide recommended for the treatment patients with grade II obesity with obesity related comorbidities and individuals with grade III obesity regardless the presence or absence of obesity related diseases [1]. As hypoglycaemia after bariatric surgery (HBS) is a rare and late complication that can occur after various types of bariatric surgery procedures, which despite being rare with the increase in popularity and number of interventions performed is becoming increasingly relevant from a clinical perspective as it affects a significant number of patients. This lead to the need of creating a platform that allows the healthcare professional to analyse not only the patient's sugar level throughout the day, but also the food ingested and symptoms experienced. With this in mind it will be necessary to interconnect the different components required retrieving the information in order to allow a more comprehensive and outreach analysis. Further on, the study carried out will be presented, which makes reference to some existing tools that could be used to develop a more efficient solution.

2.3 Freestyle Libre

Freestyle Libre® (Abbott) was chosen over its commercial competitors due to availability and proven suitability for the research task. However, the system and software were devised for distinct clinical purposes, namely for patients with diabetes, thus requiring targeted parallel analysis of the data collected.

After being introduced the uses of the system above mentioned it will be necessary to understand how it is possible to implement it according to our needs. A key part of this project will be to extract the interstitial fluid glucose (IFG) values obtained to perform the calculations of the variables that help to better understand the glycaemic variability in the study targeted population: individuals submitted to bariatric surgery and without diabetes at the time of evaluation.

In 2016, the Freestyle Libre (FSL) Flash Glucose Monitoring (FGM) System was introduced to market in Portugal, promising to revolutionize diabetes care by providing a calibration-free and affordable device yielding added information on glucose patterns that rapidly proved to be crucial

for personalized diabetes treatment. The system is composed of a sensor that is placed in the back of the arm, which measures only 35 mm by 5 mm, and lasts for 14 days, providing interstitial glucose measurements every 15 minutes and additional estimates whenever requested by the user; and a reader, which stores the data collected by the sensor and additional information introduced by the user (e.g. meals, insulin doses administered). Additionally, the reader built-in software provides glucose trends through previously established algorithms for patients with diabetes[3]. Each sensor has an inner memory of only 8-hour data, thus timely transfer of data to the reader scanning the sensor, is crucial for data retrieval, storage and accessibility. The values stored by the reader can then be downloaded to the Libre View cloud, which stores them, allowing them to be shared with the health professional. More recently the mobile application "Freestyle LibreLink - PT" was made available in Portugal, allowing the equipment to be provided with the functions made available by the sensor. This application is only available in types of equipment fitted with NFC. At the end of 2020, the FreeStyle Libre 2 arrived in Portugal with new features that allow those who use it to have a better view of their IFG levels [4]. This new equipment, with its glycaemic range from 40 to 500 mg/dL[3], presents an even better accuracy of the glucose values obtained, particularly in the low glucose range, when compared to its predecessor. It also provides the possibility of setting hypoglycaemia (low glucose levels) or hyperglycaemia (high glucose levels) alarms. This is made possible because the equipment uses Bluetooth to communicate with the sensor sending it the values in real-time.

2.3.1 xDrip+

In an attempt to fill the need for a mobile application capable of presenting the values automatically without having to resort to the reader, xDrip+ appeared, among other alternatives. xDrip+ is an unofficial and independent open-source application available for Android that stores and processes the information received from the sensor [5, 6]. This application allows a direct connection as far as Freestyle Libre 2 is concerned, but when it comes to the first version of the sensor you have to rely on another device that reads the sensor's values and sends them to the device. This application, in addition to displaying the values in graphs, also performs a predictive simulation model, enabling the user to obtain an estimate of the glucose value. xDrip+ has its own algorithm which consists of approximating a linear function using the weighted least squares regression method by weighting the calibration points based on their value.

2.4 Glycaemic Variability

To understand the triggers for hypoglycaemia, it is necessary to understand the factors that impact on glycaemic variability in the targeted population, i.e., individuals previously submitted to bariatric surgery. In [7], it is mentioned that glycaemic variability is a well-established risk factor for hypoglycaemia and is the end result of the individuals´ hormonal response to meal stimuli, i.e., frequent peaks of high and low glycaemic values that characterize the unstable glycaemic profile. For generations now, glycosylated haemoglobin A1c has been used as a surrogate marker

of medium to long-term glycaemic control. Our group has previously proposed and validated measures for assessing glycaemic variability in our targeted patient population [1]. There are a large number of such metrics that can be used to assess glycaemic variability in this specific patient population, such as interquartile range (IQR), mean of daily differences (MODD), and continuous overlapping net glycaemic action over an n-hour period (CONGA1). These are just some of the metrics used to monitor quantitatively and qualitatively with the most pertinent to the objective being the following:

- Mean Absolute Glucose change (MAG change), Useful for assessing the variability of glycaemic value by time interval, is described by equation (2.1)

$$MAG\ change = \frac{\sum_{t=1}^{t} |G_t - G_{t-1}|}{\Delta t} \tag{2.1}$$

  where G_t is the IFG record at time t presented in mmol/l and Δt is the time interval in hours.

- CONGA1, used to describe intraday variability, being very useful in identifying patterns and evaluating fluctuations. As IFG records are 15 min apart, CONGA1 accounts overlapping 60-min periods. It is possible to obtain CONGA1 applying equations (2.2), (2.3), and (2.4)

$$CONGA1 = \sqrt{\frac{\sum_{t=1}^{t_k} (D_t - \overline{D})^2}{k-1}},\ t > 60 \tag{2.2}$$

$$\overline{D} = \frac{\sum_{t=1}^{t_k} D_t}{k} \tag{2.3}$$

$$G_t = G_t - G_{t-60} \tag{2.4}$$

  where G_t is the IFG record at time t presented in mmol/l, and k is the number of IFG records where there is an IFG record 1h (60 min) before.

- MODD, used to describe interday variability, and is very useful in identifying patterns, especially in conjunction with therapeutic recording, and can be described by equation (2.5)

$$MODD = \frac{1}{s} \sum_{t=t_1}^{t_s} |G_t - G_{t-1440}|,\ t > 1440 \tag{2.5}$$

  where G_t is the IFG record at time t presented in mmol/l, and s is the number of IFG records where there is an IFG record 24h (1440 min) before.

- $LBGI_{FGM}GT$ and $HBGI_{FGM}GT$, are risk indexes for predicting deviations towards low ($LBGI_{FGM}GT$) and high ($HBGI_{FGM}GT$) glucose values (adjusted *). They can be represented by the equations (2.6), (2.7), (2.8), (2.9), and (2.10)

$$LBGI_{FGM}GT = \frac{1}{N}\sum_{i=1}^{n} rl(G_i) \tag{2.6}$$

$$HBGI_{FGM}GT = \frac{1}{N}\sum_{i=1}^{n} rh(G_i) \tag{2.7}$$

$$rl(G_t) = \begin{cases} 10 \times f(G_t)^2 & \text{iff } (G_t) < 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.8}$$

$$rh(G_t) = \begin{cases} 10 \times f(G_t)^2 & \text{iff } (G_t) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$

$$f(G_t) = 30.3082 \times |\ln(G_t)|^{0.1356} - 1.0726 \tag{2.10}$$

  where G_t is the IFG record at time t presented in mmol/l, and N is the total number of IFG records.

- Average Daily Risk Range (adjusted) ($ADRR_{FGM}GT$), Cumulative variable that results from the sum of the maximum LBGI and HGBI values in each day of IFG evaluation (adjusted*). Useful for determining blood glucose variability with defined and variable time representation. It can be described by the equations (2.11), (2.12), and (2.13)

$$ADRR_{FGM}GT = \frac{1}{M}\sum_{i=1}^{m}[LR^i + HR^i],\ i = 1,2,...M \tag{2.11}$$

$$LR^i = max[rl(G_1^i),...,rl(G_t^i)] \tag{2.12}$$

$$HR^i = max\left[rh(G_1^i),...,rh(G_t^i)\right] \tag{2.13}$$

  where i represents a day, G_t is the IFG record at time t presented in mmol/l, and M is the number of IFG monitorization.

By registering meals through photography, nutrient composition analysis by image processing arises as the next step towards automatization and a tangible goal. By conducting a search on

*Classical formulae were "adjusted" to establish risk in glucose-tolerant individuals (target range: 70–140 mg/dl; 3.9–7.8 mmol/l) monitored with FreeStyle Libre, Abbott Diabetes Care, Maidenhead, UK (device range: 40–500 mg/dl; 2.2–27.8 mmol/l)

this topic helped to understand which methods were already developed, giving more importance to those that show a better performance in automatic food recognition and volume estimation. In [8] it is recognised that food recognition is a complex task encompassing several domain-specific challenges. Several factors complicate this task, such as the absence of spatial arrangement information that could serve as an aid, as in the human body the spatial relationship between body parts. There is also the quality of the image that depends not only on the camera of the smartphone, but also on the light qualities of the environment which it is taken. There is also the quality of the image that depends not only on the camera of the mobile phone but also on the environment light qualities in which it is taken, among other bottlenecks and difficulties that could hamper the achievement of this goal.

Similarly, manually recording the volume of food is a tedious process in addition to involving a human error rate of up to 30%. Previous studies show that using the mobile phone's camera to estimate food volume increases the accuracy of calorie estimation. This study provides an in-depth insight into computer vision-based approaches. It focuses on both traditional and deep learning methodologies for feature extraction and classification methods used for food image recognition. Different image datasets are also explored and compared. With this, it was possible to determine that the performance of any methodology depends on the dataset used. They concluded that among the most used datasets, two of them were Japanese food and a third was American fast food, which contained images taken from various websites. Despite the fact that none of these are applications are likely to be useful to the general European population, by conducting this study, it was found that deep visual techniques replaced traditional machine learning methodologies for food image recognition. The conclusion was reached that 38.1% of the datasets are generic, including multi-cultural dishes. Of the analysed applications, 46.2% implemented convolutional neural networks (CNN) for food recognition, while 45.2% of mobile applications implemented traditional methods for feature extraction. To complement, 34.5% of the techniques for volume estimation need multiple images, while the remaining methods use only one image to estimate the food volume.

2.5 Proposed Solution

Firstly, since we will be using the FreeStyle Libre 2, we will need to make a connection to the sensor to be able to read and interpret the values recorded by it and, as this is the second version of the sensor, the connection will be achieved directly. Afterwards, it will be necessary to create an interface for the user to be able to photographically register his food, as well as the symptoms experienced by him when registering the meal. This photograph of the food will have to be processed to be able to be stored in the database. To store all the necessary information of each patient a local database will be created which will store not only this information but also the symptoms registered by the user, to be then uploaded to a database in a server later enable the health professional to analyse the information.

Chapter 3

Aplication Desgin

The state of the art that was conducted and explained in the previous chapter allowed the perception of the need for an application that would serve as a tool to assist in the study and prevention of the PBH phenomenon.

This chapter will also serve to expose some of the discussions and case studies analysed during the development of this project.

3.1 Design Considerations

In order to be able to achieve the functionalities desired by the professionals working to address PBH, it was necessary to start by obtaining a list of those functions. With their help and cooperation a consensus was reached on the need for the following functionalities:

- Registration of the user, with Name, Email, username and password;
- Logging in with the credentials, if this user has been previously registered;
- Recording of the meals taken by the user throughout the day, having as parameters the date and time of that meal, the user's glucose level before the meal started, a brief description of the food eaten, with the possibility of adding, alongside the description, more detailed information on each food item accompanied by the quantity of each one together with a photo of what was eaten and also, if necessary, indicating whether any medication was taken during the meal;
- Registry of symptoms that the user felt throughout the day, being necessary to indicate the date and time of their beginning, a description of what was experienced, the end time or their duration, besides the description, it is possible to indicate individually the symptoms, and also indicate if their resolution was spontaneous or if the user had to resort to some food or medication, for example;
- Possibility of editing the meals and symptoms previously registered;

- Showing of meals or symptoms, depending on the screen, grouped by day and ordered by time.

With the functionalities described above, it was pertinent to move on to the design phase of the various screens that would make up the application. To sketch the prototype it was decided to use Figma [9], which is a web application for prototyping design projects. Figma was chosen because it is a web-based tool focused on the development of graphic design systems, focusing on the development of the graphic interface together with the user interaction with this interface. Figma allows the user to create links between screens, observe the behaviour of buttons and actions, which gives a better insight into the application's behaviour, thus helping to design its flow. This tool also enables collaborative development in real time with other users.

There are several alternatives to Figma, but after a brief research it was the tool that was chosen. Of the possible alternatives for the prototype design, the decision was between Figma and Adobe XD [10], which is a more popular option among users. To use the latter it would be necessary to download the software, which in itself already presents a negative point in relation to Figma, which does not require any download because it is a web tool. The most important factor in this choice was the easiest intuitiveness, which of the tools would take less time to learn to use and, once again, Figma took the best because, with the short research carried out it was perceived that the use of Adobe XD would take more time to learn [11].

Bearing in mind the intended requirements, the prototype of the application present in Fig. 3.1, 3.2, 3.3 and 3.4 was elaborated. Considering this only a prototype and a guide for the application design, the development process was triggered.

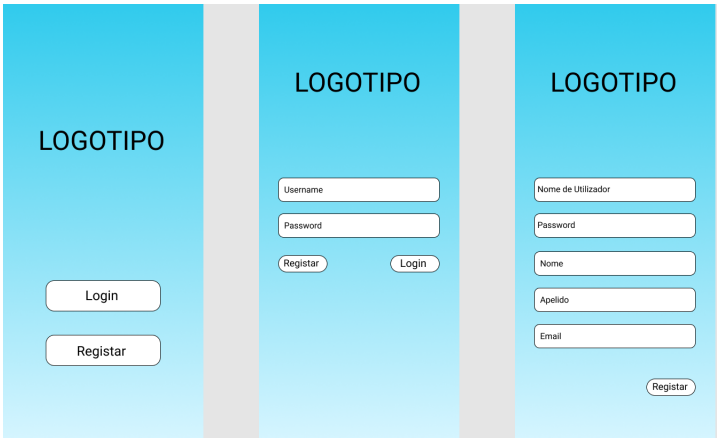


Figure 3.1: Prototype 1

Through the prototype shown above, it was possible to arrive at the architecture and flow of the application represented in Fig. 3.5 and 3.6. With this, it is possible to understand the behaviour required for the application to work in the most optimal way to be able to deliver the best user experience to the user.

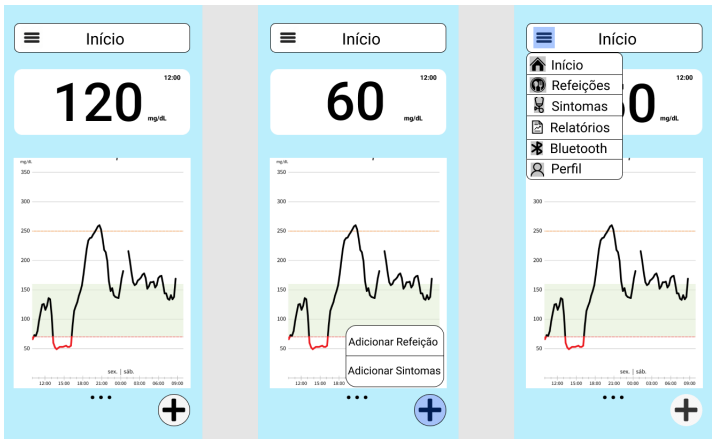


Figure 3.2: Prototype 2

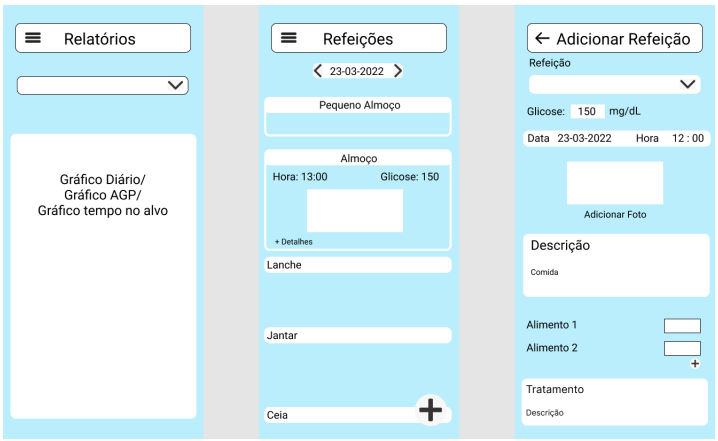


Figure 3.3: Prototype 3

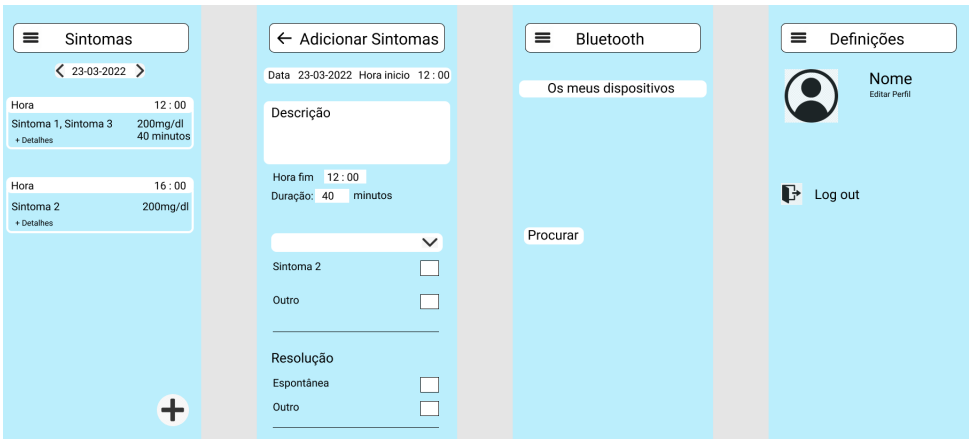


Figure 3.4: Prototype 4

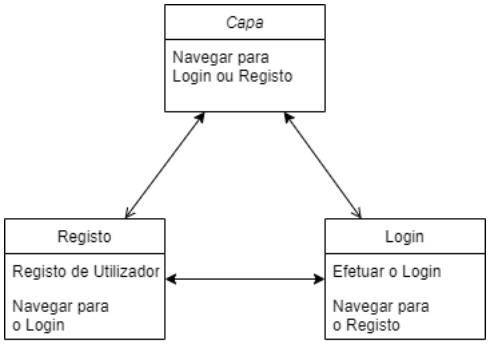


Figure 3.5: Application Login Flow

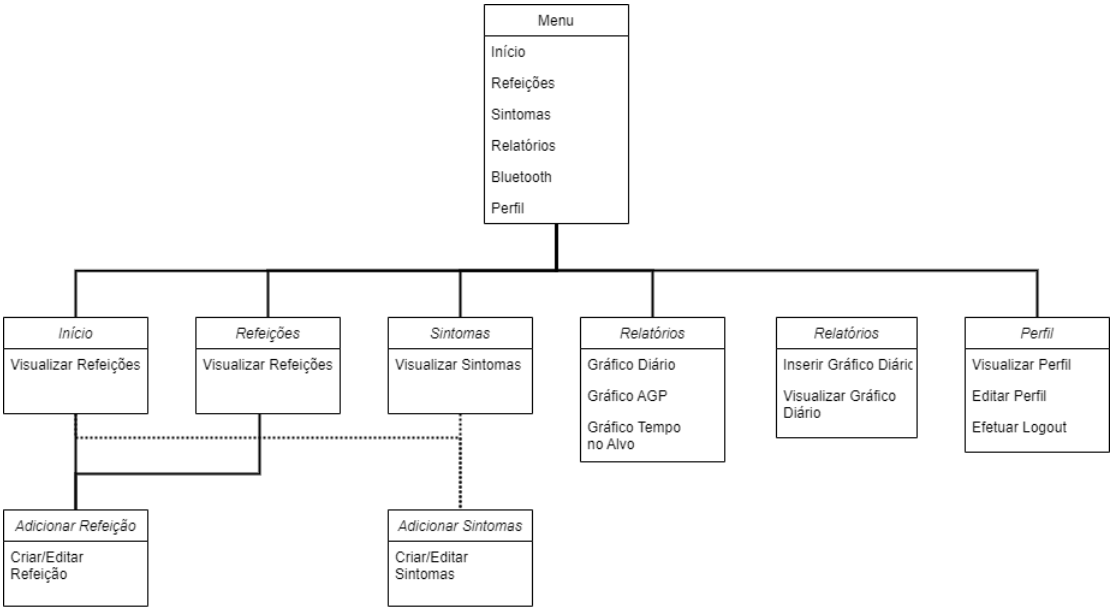


Figure 3.6: Application Flow

3.2 Development

Choosing the development environment and language for the application was an important step. Taking into account that we had in mind that the application would be for Android and for iOS, distinct operating systems, it would be fundamental to choose accordingly. With that in mind, Flutter [12] was used to develop the application's graphic interface. Flutter is a framework created by Google based on the Dart programming language that allows the creation of natively compiled applications for various operating systems, including those intended for the project.

Another excellent option for the language and platform would be React Native [13], developed by Facebook, which also enables development on various platforms. React Native, being older than Flutter, has a more significant number of packages and documentation, which did not spark much interest. The two alternatives presented have a slight difference in the language used, with React Native using a language based on JavaScript, which makes it a dynamic script, while Flutter allows not only using a dynamic script but also a static one. This means that Flutter does type checking both at run-time and compile-time, whereas React Native only does type checking at run-time, which makes the code produced by Flutter more robust and poses less risk of assigning values to the wrong data types.

In the end, the choice fell on Flutter because, as also happened with Figma, it is more intuitive and has a lower learning curve than React Native. Because Flutter uses the Dart language, it provides faster compilation than JavaScript. To implement Flutter, we used Android Studio, which allows the use of emulators to test the application, as well as install the application on an Android device. With the help of Hot Reloading provided by Flutter, it is possible to test the changes made immediately since using this tool only the changed code is compiled instead of the entire application. Flutter provides a wide choice of pre-made libraries well documented on how to use them, developed by Google and the Flutter community. Among these packages are some that are needed in the course of developing the project application. As already mentioned, Flutter provides the app built natively for Android and iOS without having to develop them separately. However, some disadvantages to using Flutter come from its rapid growth, leaving some libraries outdated as functions and methods change. To combat this problem, Flutter has made a migration guide that aims to help developers migrate from old to new versions.

3.3 Backend

For the backend of the application, it was decided to develop a local database capable of storing the meals and symptoms entered by the user to possibly access the records faster. The use of a local database meets the need for greater availability of certain records in an almost immediate way to give the user the possibility to understand their glycaemic variability better. Therefore, it is intended to have stored in that database the meals and symptoms for about a week so as not to take up excess space.

With this database, it is necessary to have a server capable of managing all kinds of data, users and the records made by each. For this, we want the server to have a database capable of archiving a large amount of information. The server would then be responsible for registering users and the records associated with each of them.

For the development of the server, there was the possibility of using already existing services, such as cloud-based services. Cloud-Based means that the service is managed by another entity responsible for the server and all related traffic. By using an externally managed server, you would save on server development as the server does not need a very detailed configuration. With the available time being tight to perform a deep research on the various services already available and with the need for further customization of the server with regards to requests made and communication with the database, the decision fell on the choice to develop a server instead of the available cloud-based ones.

Since the choice was to develop a server from scratch, it would be necessary to consider which development environment to use and, among the various options available, the project's supervisor advised me to use Node.js [14] to develop the server. Node.js is a backend JavaScript runtime environment, cross-platform and open source and was designed to build scalable network applications. Being an asynchronous event-driven JavaScript runtime, Node.js is designed to build scalable network applications, making it lightweight and efficient compared to threaded servers. This environment enables the implementation of frontend and backend applications without the need to change language and environment due to the possibility of creating everything with JavaScript. The main task of a web application server is to respond to and provide data from client requests. Node.js has a built-in HTTP module, which allows you to create an HTTP server that listens on the server ports waiting for client requests and then sends the response to the client.

Also advised by the project supervisor was the Koa framework, which aims to create a more robust foundation for web applications while being smaller. According to the Koa's website [15], Koa was designed by the team behind Express, the NodeJs Foundation [16], and taking advantage of async functions allows for less use of callbacks and improved error handling. Koa allows a more detailed configuration so that developers can create the application without any restrictions, and some of the aspects that make this framework a good choice are:

- Counting with the most recent and modern version of JavaScript, always keeping up with it; Being one of the lightest existing, it allows the user to install only the necessary dependencies, thus occupying less storage space;

- As, at installation, it only installs what is necessary for the minimum operation. It does not present any middleware present, allowing the user to create or choose from among the existing ones created by the community without presenting any restrictions. This allows developers to be more open about what they want the application's requests and responses to behave like.

Due to the use of Koa, a better management of the middleware would be possible, which would help, since the server also needs a database responsible for storing the user's information, as well as for registering his meals and symptoms.

3.3.1 Database

As previously mentioned, it was necessary to have a local database in order to provide a better user experience. For simplicity, it was decided to have both the local database and the server database developed in SQL. The use of a local database would be used to improve the user experience. This database would be filled at login with the data from the last week, and only if necessary the server would be used again to obtain the missing information. The idea of only filling a not too long time interval is not to occupy too much local memory with information that is not considered immediately necessary.

The two databases created are almost identical, with the server database consisting of one more table than the local one because it is necessary to store data from the various users in addition to the shared tables in the databases. The shared tables are the tables for storing meal data, another for storing symptom data, and a table for storing the photo of the daily blood glucose graph. Starting with the local database, as mentioned before, this is made up of the tables meals, symptoms, and photo, each having the indispensable columns for each to be as complete as possible regarding information. The following diagram represents the database present on the server:

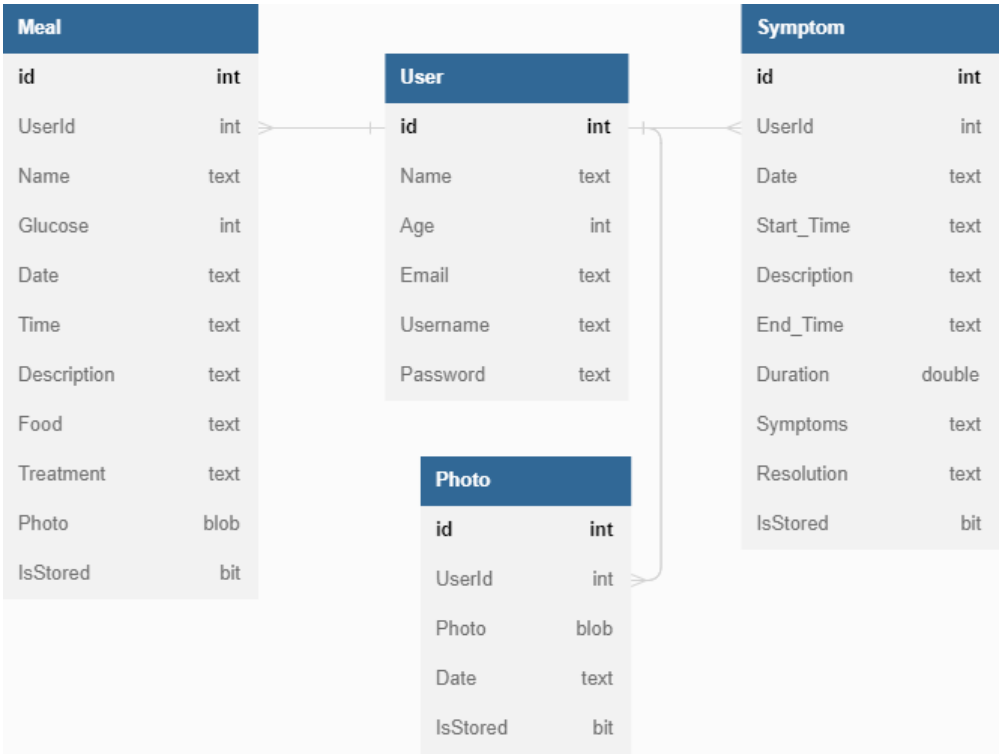


Figure 3.7: Database diagram

3.3.2 Authentication

In the current days where security is taken very seriously, it had to be the same here, and since the authentication was done from scratch and no third party was used, some steps had to be taken in order to make the passwords secure. For this application, a simple challenge-response protocol between the application and the server was chosen in conjunction with the project supervisor. This requires that, at the time of registration, a slow hash *H1* is stored in the database, using, in this case, the Argon2 function, created based on the password entered and stored in place of the password that the user entered.

To be able to start the login process, the user begins by entering the data on the application's login screen. The application sends to the server the username entered and waits for a response from the server. The server's response will return a random string generated by it, which will also be saved for later use. The application receives this string generated by the server and, concatenating it with the *H1* obtained from the entered password, uses it to calculate a second hash achieved using the *SHA-256* function. After obtaining the result of the function, it is sent to the server that, like the application, will calculate the new hash using the function previously used, the string previously generated by the server and the *H1* stored in the database. With the second hash calculated, a comparison is made on the server between the value obtained and the value sent by the application, transmitting to the application the result of this comparison, indicating whether the login was successful.

Finally, and according to the result of the communication with the server, the application will take the necessary action, indicating to the user that the password is wrong or proceeding to the application.

Chapter 4

System Structure and Screens Created

The main objective of this chapter is to explain in great detail the system's structure in the developed solution. Firstly, the system will be presented in general terms, without going into great detail. Next, the various screens created using Flutter will be presented, explaining their purpose and the functions created for better functioning. Initially, the elements that were created to help develop the screens will be discussed since they are frequently used in the development of the application. Finally, all the screens that make up the application will be described in detail.

4.1 Application Structure

The designed application is structured using three main components: the mobile application developed in Flutter, the server responsible for receiving the requests made by the user and communicating with the database, and the local database. Furthermore, a web platform will be created for viewing and analysing all the data entered by the user in the mobile application, aiming at its use by health professionals. Taking into account what was explained above, Fig. 4.1 highlights the system that is intended to achieve.

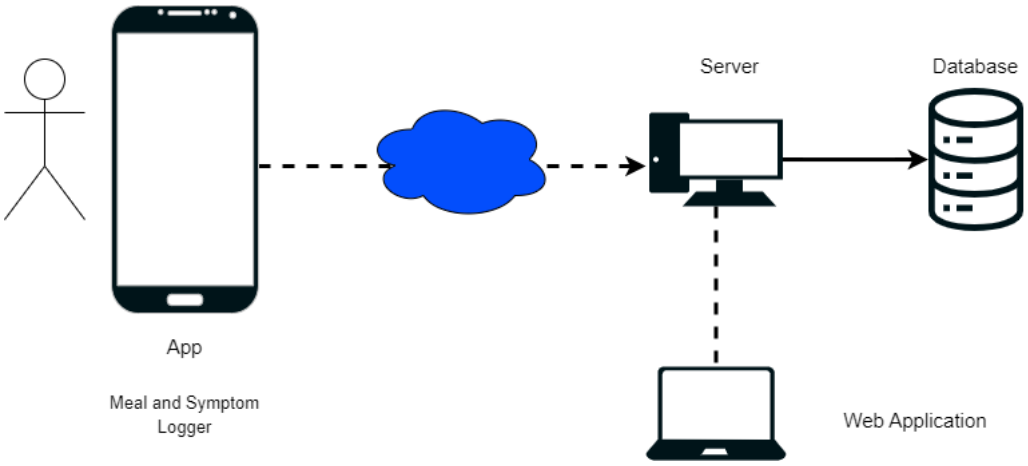


Figure 4.1: System Structure

The main goal of this project was to develop a mobile application capable of meeting the proposed requirements. To make the registration of the user through the application, make the registration of the user in the database. After the registration is made, it will be possible for the user to start recording their meals and symptoms throughout each day. This information is initially stored in the local database to improve the user experience since the most recent information is stored locally, dispenses with the communication with the server and thus has a faster viewing of content. The information initially stored locally will be sent to the server for further registration of all data. To send this information, it is necessary to verify the existence of communication with the server and only if this is verified it is then sent to it. At the Login moment, the local database will be empty, and it is necessary to fill it with the information on the server. It is possible that the user has not yet registered any meals or symptoms, so the data transfer does not exist. To start this transfer of information, connectivity to the server is verified, as described above, and if it is verified, the information transfer is performed.

4.2 Screens and Elements Created

The application has multiple screens, each with its specific purpose and corresponding design. In this section, we will explain how they were implemented, their purpose and the various adjacent functions necessary for their operation.The first three screens presented, which are the cover page, the login page and the registration page have as background a linear colour gradient scheme achieved through the *LinearGradient*, starting in a kind of *Picton Blue* (*#31CBED*) and ending in a *Mabel* (*#D5F5FF*), both belonging to the blue colour family. For the remaining screens, we chose a colour from the blue family but aimed to obtain a more neutral and appealing colour to the user, falling on the *French Pass* (*#BBEEFC*).

4.2.1 Standard Pages

4.2.1.1 Front Page

Regarding the opening screen, it was intended to have an appealing design, not too crowded, as previously presented in the prototype and, to meet the proposed, the goal would be to replicate the colour obtained in the prototype to use as background colour and, as previously mentioned, *LinearGradient* was used and satisfied the intended requirements. Concerning the front page, it was aimed to have only the application's logo and two buttons, one to direct the user to the Login page and the other to the registration page. Through this, we reached the result seen in Fig. 4.2.

The buttons are the same, changing only the text inside as they serve different purposes. Both were built using the *TextButton* widget since it allows the developer to insert text inside the button, which was intended for this situation. The widget mentioned above was placed inside a *Container*, a kind of box that offers great customisation such as borders, colour, and contour, which were used in this context to obtain the buttons shown in Fig. 4.2.

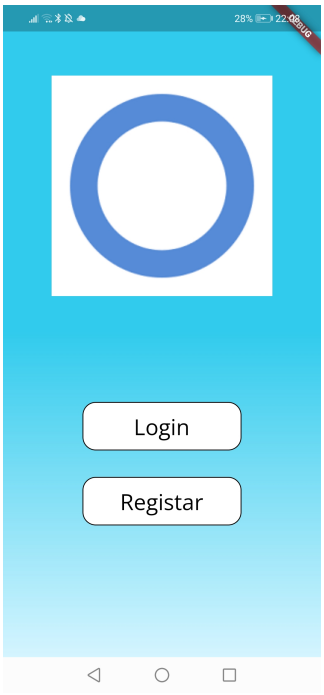


Figure 4.2: Cover

4.2.1.2 Login Page

The login page is standard in almost all the applications on the market due to the importance of saving each user's data in order to be able to access the account and all the information stored by the user. As usual in a login page, it is necessary to have a field for the user name or e-mail as well as a field to insert the password so that, if the username/password combination is correct, the user can access the application and make the records that will be saved. As previously stated, the colour gradient will be used as the intended background. Above the form fields, it is intended to have the logo, as it is on the front page, but on a smaller scale. The form fields were made using *TextFormField*, Flutter's class that allows the user to insert data of several types that later allows the application to manage and treat the data in the necessary way, depending a lot on what is intended on the screen. In the case of this screen, the objective is to allow the user to insert the necessary data so that he can log into his account. The *TextFormField* are each inserted inside a *Container* to achieve better customisation, which would not be possible using only the decoration present in the *TextFormField*.

With the encapsulation inside a *Container*, it is possible to customise the exterior, the surrounding box, so to speak, of the form field. Concerning the verification of what the user inserted, this checking was performed by resorting to *TextFormField* functionalities which, through the analysis of what was inserted in it, presents the user with a feedback message in relation to what was inputted by the user. Besides that, as it is the login page, it was pertinent to put a Feedback message about the login itself, which printed a success message, in case it happened, or an error message with whatever the error found.

Besides the form fields, there are still two buttons that are necessary on a login page being them the Login button, which is responsible for evaluating the information that the user introduced in the fields and that is there at the moment the button is pressed, and a button that directs the user to the register page, which presents the text *Registar* inside. This second button serves in case the user has somehow made a mistake on the previous page and wants to make a registration instead of logging in to an existing account. Another situation where there is a need to go to the registration page is when the user is informed that the username is not in the database and wants to register. These buttons were also inserted inside a *Container* allowing a greater personalisation, as mentioned before. The design reached, in the end, was the one seen in Fig. 4.3 that comes from the above described, with the customisation application in the containers, as stated above.

Figure 4.3: Login

4.2.1.3 Registration Page

Like the Login page, the Registration page is also standard in most applications, being always mandatory whenever the application has information regarding each user that is pertinent to keep stored and saved so that only the user can access his/her information, allowing him/her to do that from any device that logs in. Always walks alongside the Login page since they complement each other, not being able to access an account that is not previously registered. In the same way as the pages described above were built, this one also used the same colour gradient as the background. The need for inputs from the user side so that he can enter his data to proceed to his account registration was used, the same way as before, the *TextFormField*. In this case, also inserted inside a *Container* so that we can proceed with the customisation as before, but using more the properties

of the Form Field given that it would be essential the existence of a more significant validation of each field so that the user can understand where he made a mistake in the registration of the account. In order to register, only four fields are needed: the username, the password, the email and the name of the user. The first two fields were previously seen as the necessary data to log in to the application, so they need to be present in this account creation screen. Having all this described, we then arrived at the screen that can be seen in Fig. 4.4. As in the previously mentioned screen, there was a need for a registration button so that the application could gather, validate, and proceed to the user's registration. It is also necessary to return to the login page if it is verified that the existence of an account with the user and email that were inserted by the user in the registration fields, since the account already exists and only needs to log in to access the application, or else in case of a mistake by the user, in which he pressed the unwanted button on the front page.

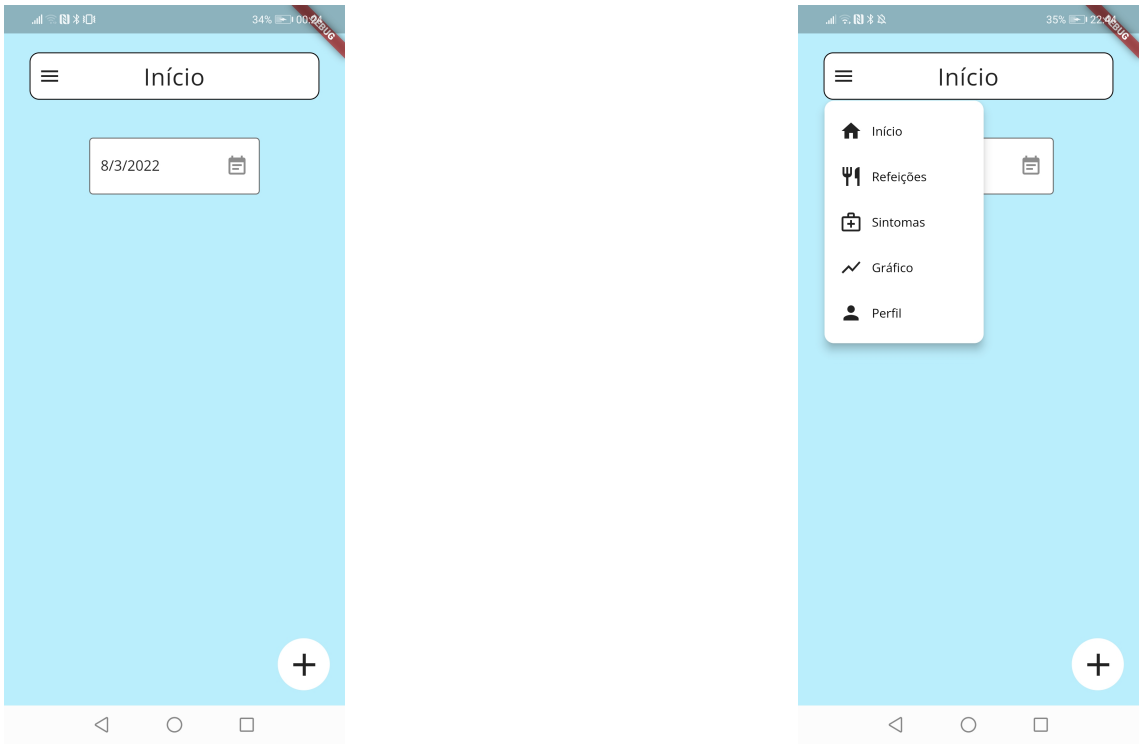
Figure 4.4: Register

4.2.2 Page Elements

When starting to develop the application, the need to create some elements that would serve in several pages arose, thus preventing code repetition and being able to reuse code. With this emerged, among other elements, the templates are exposed and explained below, presenting the thinking behind them.

4.2.2.1 App Bar

In order to meet the desired and the presented prototype, it was understood that the best way to obtain the bar with the navigation page information and the application menu would be to create



(a) Home Page

(b) Home Page with Menu

Figure 4.5: Home Page

it from scratch. Instead of using the *AppBar*, a class belonging to Flutter consisting of a toolbar that typically exposes common actions to the application. As stated in [17], the *AppBar* is usually placed in the Scaffold area, which makes the bar a fixed-height widget that is placed at the top of the screen. As we wanted the bar to be slightly below the top of the page and not the screen's full width, we then proceeded to create our *AppBar*. The goal was to have inside the bar a menu, represented by the icon, and the name of the screen in question, being necessary for the creation of our menu and, using the *PopupMenuButton*, it was possible a not very difficult implementation of the menu in question, having been more time consuming the insertion of the icon corresponding to each element of the menu. The name of the screen in question was relatively easy since it is just text, using the Text widget, we achieved the desired result. As previously done, all the elements were placed inside a *Container* that would have a width of 85% of the screen width for this purpose. As it was intended to have the widgets in a row, we used the *Row* widget, which allows having several widgets inside, forming, as the name indicates, a row, having necessary to use the Expanded widget in order to expand each element as intended. With this, we reached the result seen in Fig. 4.5, presenting the bar in a screen context.

4.2.2.2 Float Button

The next element arose from the requirement to have a button that, depending on the screen, presented the option to move on to another screen, this could be the Add Meal or Add Symptom

screen. Intending to fulfil this goal, the element *FloatButton* was created, achieving what can be seen in Fig. 4.6, a kind of *FloatingActionButton*, even being implemented on the screen as such. This element was created from scratch, having the *FloatingActionButton* as an example since it was the goal to have a similar button but one that presented a list of options when pressed. In order to obtain that, the intention would be to receive a list of elements that the user wanted to have inside the menu that was presented when pressing the button, and for that, as in the application's menu previously made, the *PopupMenuButton* was used, which would be represented by the + icon, being this widget inserted inside a *Container* so that decoration could be applied, wanting in this case to have a round button, applying a circular border to it.

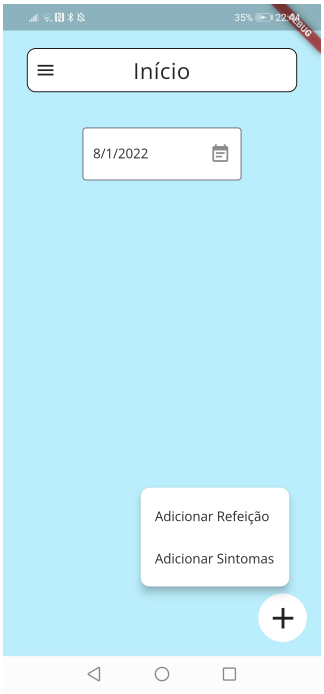


Figure 4.6: Home Page with Float Button

4.2.2.3 Meal Box

For the sampling of the information of each meal, it was intended that the user would be able to visualise some of the information that was considered essential without taking up too much space so that he could know his meals of the day.

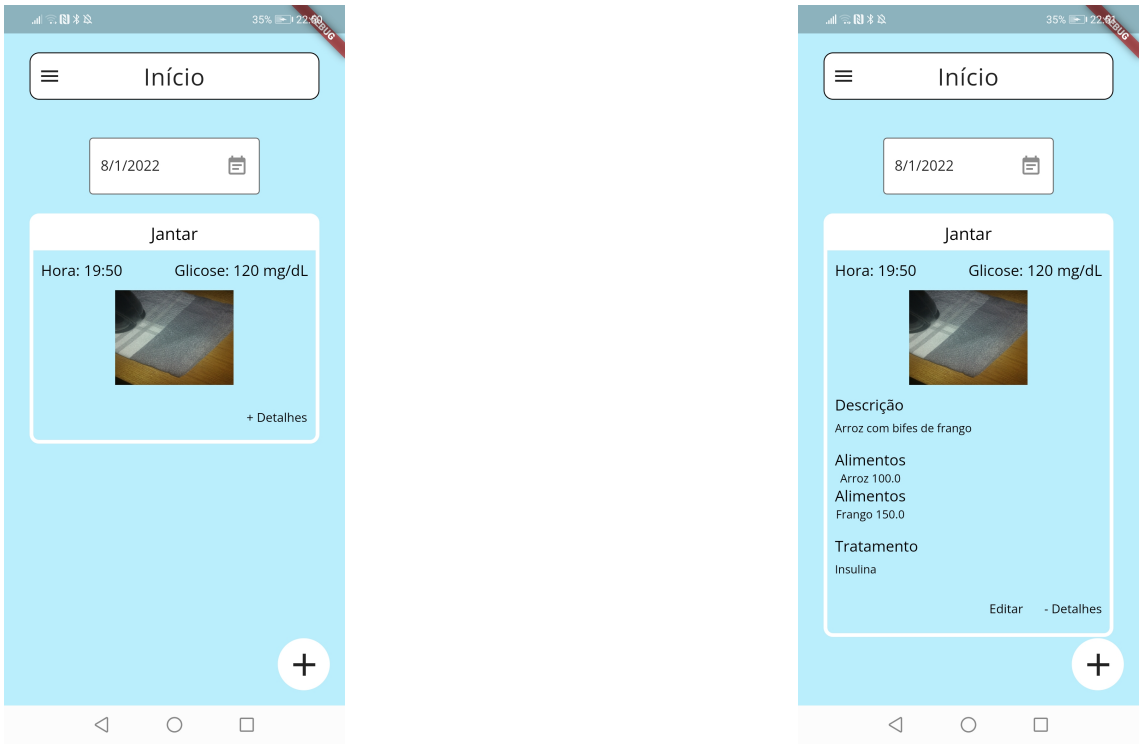
In order to achieve that, the prototype initially obtained was used as a reference, in which it was understood to show, for each meal already registered, the name of the meal, the time in which it was registered, the glucose inserted by the user, as well as the photo which represents the meal. As seen in Fig. 4.7a, all the information about the meal is inserted in a kind of box to isolate each meal so that it is easier to understand the information about each one. For this, all the necessary information was gathered inside a *Container* which, as previously explained, allows better customisation. This property was used to apply a white border rounded at the corners in

this case. Inside that *Container*, another one was used to put the title that, as it is possible to see, presents the white background so that it can be differentiated from the rest. The content of the meal, which means the data inserted by the user, is presented below the title, and, as previously mentioned, only the most important data is presented initially. It was possible to reach the possible result involving everything inside a *Column* widget, which allows better management of the several widgets inserted inside. For the first row of the column, a *Row* widget was used once it was necessary to put the hour and the glucose aligned, achieving that with its use. After this information, and to have the photograph inserted by the user, the widget Image was used, which allows loading a photo, which in this case would be a variable of the type *Uint8List*, which later on will be explained the reason for its use. As this is not the complete information of the meal, it was necessary to place a button in the right bottom corner so that the user could expand the meal box to present the complete information of the meal, the button *+Detalhes*. After pressing the button, the box expands and becomes what is possible to see in Fig. 4.7b, showing all the information about the meal. To the information previously shown, only the one that was not already there is added, using Text widgets responsible for presenting the data related to the description of the meal, the food and its quantities and the description of the treatment performed. Given that it is allowed that the user edits his meal, there is in the right inferior corner of the meal box a button that allows the user to pass to that edition of the existing content of his meal, next to the button of *-Detalhes* that passed to that name with the intention that the user, when pressing it, diminishes the meal box, hiding the dispensable information.

4.2.2.4 Symptoms Box

As well as the need to create the box mentioned above, it was also essential to have a box in the same style as the one previously created for meals, having, in this case, the objective of showing the information that the user registered regarding the symptoms.

With that purpose in mind and using the prototype presented above as an aid, we arrived at the result seen in Fig. 4.8. The box itself does not present many differences in terms of design compared to the previous one, changing only the content presented. In the section where, in the previous case, was the title of the meal, it was decided, in the case of symptoms, to present the time when the symptom was registered, presenting the text aligned to the left, while the time, which in this case would be a variable, is shown aligned to the right. In the content section, only text was shown, and the alignment was easier compared to the previous element. In order to get to what is possible to see on Fig. 4.8, the texts were grouped in a *Row*, which makes it possible to place widgets side by side, which was the goal. These rows are inserted in a *Column* widget to be aligned, with a button at the end that gives the user the opportunity to change the register previously made. Both sections previously discussed are inserted inside a *Container* to be able to apply the style as it was applied before.



(a) Meal Box

(b) Meal Box Expanded

Figure 4.7: Home Page with Meal Box

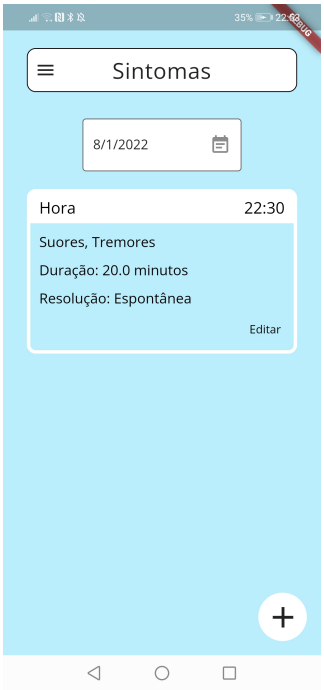


Figure 4.8: Symptoms Page with Symptoms Box

4.2.3 Application Specific Screens

In the previous subsection, the most common screens were presented, which can be found in most of the applications existing in the market, now going to the screens that are specific to this application and necessary for its operation. In a detailed way, the following screens will be introduced:

- Home;
- Meals;
- Add Meal;
- Symptoms;
- Add Symptom;
- Daily Graphic;
- Profile.

4.2.3.1 Home

As a home page, the initial idea was to show the user's blood glucose which would be taken from the *FreeStyle Libre* sensor, followed by the day's meals. Later on, the reason will be exposed, but it was impossible to obtain the values from the sensor, so we had to choose to show only the meals grouped by day, allowing the user to choose the day he wants to see. As it is possible to see in Fig. 4.5, 4.6 and 4.7, the home screen has an App Bar, which includes the name of the screen in question and the menu that allows the user to navigate through the application between the various screens. This page also has a widget that allows the user to choose the date that he intends to visualise, which was created using a *DateTimeFormField* in date-only mode. Depending on the date selected, the user will be shown the meals registered by him on that day, and, since the meals are being shown in the meal box previously mentioned, it is possible for the user to see a small summary of his meal, as well as expand the meal box and see all the details inserted by him regarding that meal, also having the possibility to edit that meal. On the home page, it is also intended that the user has the possibility to add a meal or a symptom, having used for that the *FloatButton*, an element created and previously discussed, which satisfies that need, serving the purpose for which it was created.

4.2.3.2 Add Meal

The user must be able to register the meal, and to fulfil this requirement, the "Add Meal" screen appears, allowing the user to insert the meal's data. The screen is made up of fields, some of them mandatory, which give the user the chance to use the fields for entering the details of the meal, and the final result can be observed in Fig. 4.9. For this, the screen has been created with the fields for:

- the type of meal, mandatory;
- the glucose before the meal, optional;
- the date, mandatory;
- the time, mandatory;
- a photo of the meal, optional;
- a description, mandatory;
- a section for the food and quantities, optional;
- a description of the treatment performed, optional.

Figure 4.9: Add Meal

Access to this screen is made through the element previously created, the *FloatButton*, which, when pressed, shows a menu with the options available on that screen. By pressing the "Add Meal" option in that list, the user is directed to that page where he can create his meal.

The screen was implemented within a *SingleChildScrollView*, allowing the user to scroll through the page to see all the widgets that are part of it. As on the home page, there is also a custom created top bar consisting of the title and, in this case, instead of the application menu, there is a button that allows the user to go back to the previous screen. When arriving at this screen, it has no information filled in, except for the date and time fields set with the current data. To choose the type of meal, the user must choose from those presented in a *DropdownButton*, and it is mandatory to make a choice. The following field, of optional character, allows the input of the glucose checked before the meal, followed by the date and time fields that, even though they are filled in initially, can be changed by the user in the case of a previously made meal. Following this is the photo input field, designed using a ready-made Flutter package called *image_picker* [18]. The package handles the application's connection to the device's camera and gallery, making it possible to choose a photo from the gallery or take a photo from within the application. In this field, and to allow the user to choose where the photo comes from, two buttons are presented, one to access the gallery and get the photo from there, and the other to access the camera to take the desired photo. The chosen photo can be changed even after it is uploaded to the page. Next, the user has to briefly describe his meal, often complementing the previously inserted photo, and for that to be possible it was used, inside a Container, the TextField widget that allows the user to insert text, having in this case the purpose of being the field for the description. Also, as an optional feature, the following fields complement the information entered so far. First, there is a section where it is possible to insert in more detail the food ingested, giving the user a *DropdownButton* with some options, followed by a place to insert its quantity. Later, the last field of the form is for inserting a brief description of the treatment or other information that is considered pertinent.

By pressing the button at the bottom of the page, all the fields in the form are verified in terms of the data present in them so that it is possible to insert the information into the database.

4.2.3.3 Symptoms

The purpose of this screen is to show the user the symptoms that he or she has entered for a particular day that can be chosen. The page is initially loaded with the data of the day in which it is. However, it presents a Date Picker, implemented using a *DateTimeFormField* in date mode only, so it is possible to choose the day to visualise the symptoms. If there is data to be shown, it is displayed in the symptom box mentioned above, using one for each symptom incidence. For the user to access the screen that allows a new symptom record, the *FloatButton*, previously mentioned, was placed here, with only the option to "Add Symptoms" in the menu. It should also be noted that this screen contains the App Bar created and previously mentioned, showing the application's screen name and navigation menu.

With this previously mentioned, what can be seen in Fig. 4.8 emerged, which in a simplistic way presents the necessary information to the user.

4.2.3.4 Add Symptoms

This screen is intended to allow the user to record symptoms by filling in the fields that exist here. As in the "Add Meal" page, this page is also reached through the *FloatButton* on both the home page and the previously mentioned symptom page. The result of this page can be seen in Fig. 4.10, and each field here will be explained further. Some fields on this screen are mandatory to allow a better understanding when analysing the registered symptoms. The existing fields are to be filled in with:

- the date and time, mandatory;
- a description of the experienced symptoms, mandatory;
- the end time, mandatory;
- the duration, mandatory;
- a box to add symptoms, mandatory;
- a box to explain the resolution of the symptoms, optional.

The various widgets on this screen are implemented within a *SingleChildScrollView* to allow the user to scroll through all the page contents. As with all pages in the application, it also had to have the bar that displays the screen's name but with a button that allows the user to go back to the previous page instead of the application's menu. Next, and starting the form fields, we have the initial date and time fields, which are each one of them a *DateTimeFormField* configured to be one in date mode and the other in time mode, respectively, being initially filled with the values of the time zone where the device is. After this field and to enter the description, there is a container with a *TextField* inside so that the user can enter text. After this, there is another *DateTimeFormField* to enter the end time of the symptoms, working together with the next widget, a *TextField*, so that the user can enter the duration of the symptoms in minutes. These last two widgets are interconnected because by filling in only one of them, the application is prepared to fill in the value of the other when the form is submitted because the duration is the difference from the initial time to the final time, thus calculating this if the final time is entered or using the duration entered by the user to calculate the final time. In order to facilitate the work of the user of the application in the introduction of symptoms, there is a *DropdownButton* inside a container with a list of the most common symptoms in hypoglycemia, being possible to add more than one symptom using the "+" button below the drop down, and the user also can insert in a *TextField* a symptom not present in the list. Finally, there is a box where the user can indicate how he solved the symptoms, with the possibility of activating the Checkbox or filling the *TextField* with another reason.

Pressing the button at the bottom of the page activates the process of registering the symptoms in the database, but not before checking all the fields to see if the mandatory fields are filled in and if the data in them are in accordance with what is needed.

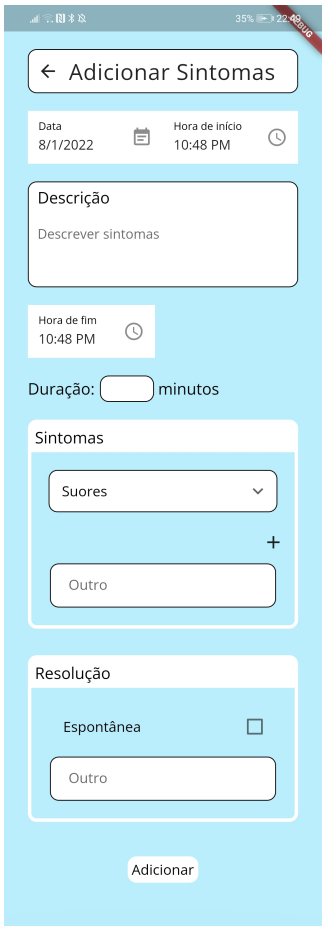


Figure 4.10: Add Symptom

4.2.3.5 Daily Graph

Initially, this page was intended to be a search interface for Bluetooth devices so that a connection could be established with the Freestyle Libre 2. Through this connection, the intention was to obtain the glucose values recorded by the sensor. In order to establish that connection, the *nfc_manager* package was used since that Bluetooth connection between the device and the sensor would only be possible by taking an NFC reading of the sensor to obtain the precise data for the connection. Since, during the connection tests, only the NFC tag ID could be obtained, the solution had to go through another alternative. As mentioned in X, such a connection is possible using a modified application, and due to limited time, it was not possible to explore this aspect. That being said, the work plan had to be changed, now having an interface where the user can insert a photo, where ideally that photo should be the graph generated by the *LibreLink* application, an application capable of collecting the sensor data generating a daily graph of the blood glucose values. With the photo of the user's daily blood glucose graph, it is possible to compare the time of the meal previously recorded with the blood glucose that can be seen on that graph, which is a great help for the professional to analyse the trends of blood sugar, taking into account what was ingested by the patient.

The solution that can be seen in Fig. 4.11 is not the ideal solution, but it was the one we opted for since it is not too far from the intended solution, since it is possible to analyse the time of the meal and what effects it had on the blood glucose of the user based on that day's graph.

Figure 4.11: Daily Graph

This screen also has the *AppBar* already presented, containing the screen's name and the application's menu, having just below that a *DateTimeFormField*, configured in date mode so that it is possible for the user to change the day he wants to visualise the inserted photo, in case there is any, having the possibility to change it. Next, there is a *Container* that shows the photo, if it exists, giving the user visual feedback if it does not exist, and the photo can be inserted using the *image_picker* package mentioned before. For the user to be able to place his photo, there are two buttons so that he can choose from where he wants to upload his photo, having as alternatives access to his gallery and choose a photo from there or access the device's camera to be able to take the photo at the moment. Finally, there is a button for the user to upload the inserted photo to the database to save it for later access, and it is also possible for the user to change the saved photo.

4.2.3.6 Profile

On this last page of the application, it was intended to have more options than the existing ones, but with the lack of time, it was only allowed to implement those considered most important. That said, this page has the AppBar with the menu just like all the others covered so far, followed by a section with the user's profile picture with the name beside it. Below the name is a button that allows the user to change the login information, which, so far, is the only information available. There is also a Log out button so the user can change the account they are logged into or just log

out. It is left the possibility of putting more buttons in this list that for now has only one because the button is inserted in a Column widget to be easier to implement more features. The initial components are inserted inside a Row widget, so it would be possible to have the photo side by side with the name. For the user photo, a CircleAvatar was used to show the circular profile photo to make it more appealing. With this, we arrived at the result that can be seen in Fig. 4.12.

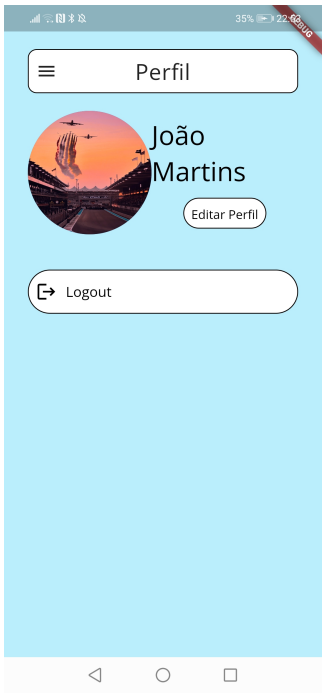


Figure 4.12: Profile

Chapter 5

Application and Server Implementation

The objective of this chapter is to explain the architecture used in the application so that it would become fluid and efficient. Initially, reference is made to the chosen architecture, followed by an explanation of how it is used in the screens so that they can access the same information. After that, a reference will be made to how the local database communicates and interacts with the existing screens. Finally, the Web server will be discussed, explaining the reasons that support the choice made and how it integrates with the application.

5.1 Application Architecture

Among the existing alternatives for the architecture, *Redux* was chosen because, in this case, together with the advisor, we considered it to be the most suitable option. *Redux* is a state control architecture library that manages the application's point of situation in a unidirectional way, which facilitates the application's development, making it easier to test and maintain. *Redux* is composed of a Store that stores the State object that represents the state of the application, application events called Actions and the Reducer that updates the Store with the new state, depending on the Action it receives. *Redux* uses the Actions to alter the State since it is immutable, which allows for greater security because in order to change any value within the State, it is necessary to replace it with a new one, updated with the new values.

To make possible the use of *Redux* in Flutter, we used the *flutter_redux* package [19], which provides a set of tools that allow the consumption of the Store to build Widgets in Flutter, being these widgets:

- *StoreProvider*, which is the base widget, that means that it will make the Store available for all its descendants;
- *StoreBuilder*, a descendant widget that receives the Store from the *StoreProvider* and passes it to the widget builder;

- *StoreConnector*, a descendant widget that receives the Store from the closest *StoreProvider* and transforms it into a *ViewModel* to be used in a widget builder. Once the widget contains the *ViewModel*, it is automatically rebuilt if there is a change in the Store.

As previously mentioned, *Redux* contains the State of the Application that cannot be changed but replaced by a new one, and for that to be possible, it is necessary to use Actions that send the necessary information to Reducer so that it can process it and proceed to create a new State with the desired information. In order for the Reducer to receive the necessary information to update the state, the dispatch method is called on the screen, which communicates to the state the Action to take. In this case, it was pertinent to create the state with the elements considered essential to the operation of the application, these elements being represented in Fig. 5.1.

State	
Refeicao	refeicao
Sintoma	sintoma
nome_ref	text
nome_sint	text
nome_alim	text
qtd_alim	int
alim_list	alimento
sint_list	text
refeicao_list	refeicao
foto	foto
userId	int

Figure 5.1: State

With the State duly created, it was fundamental to understand what the necessary Actions would be so that it would be possible to update the State whenever necessary. For that, it was also necessary to create the Reducer functions so that it could create a new state with the information from the Action and the current state, changing only the desired data. With this said, it is possible to observe in Fig. 5.2 the behaviour of the architecture as described. After some development, it was possible to obtain the functions that could update the state in the intended way. Later on, we will explain how this architecture is integrated in the application, presenting some concrete examples of the use of the Store and the use of Actions to update its state.

5.2 Local Database

In order to have information available to show on the meal and symptom screens, it is necessary to access the database both to read the data stored in it and to register the information that the user

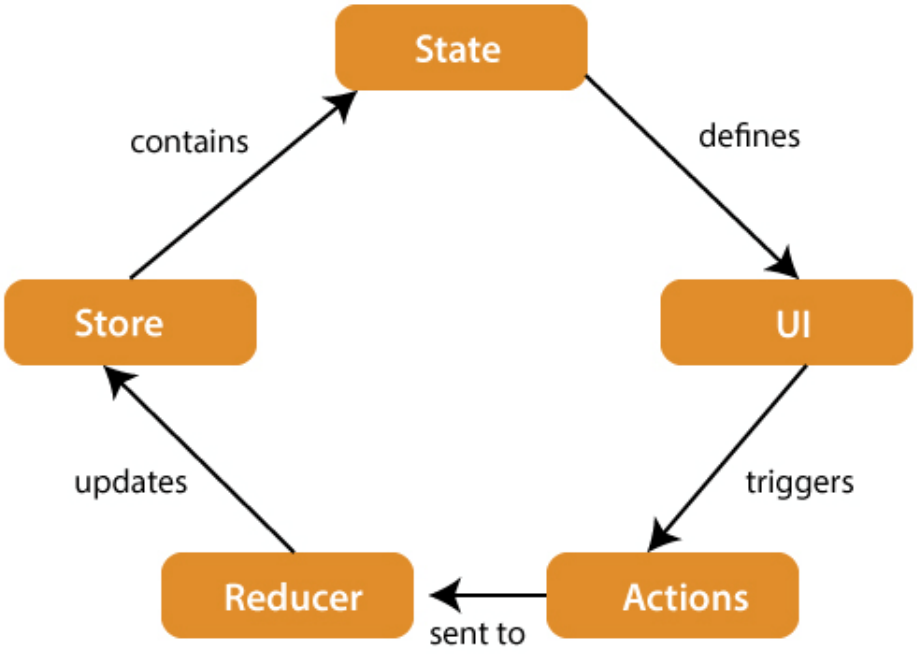


Figure 5.2: Redux

provides to the application. Since the database was developed in SQL, it was decided to opt for the sqflite package, which is an SQL plugin for Flutter that provides help when it comes to executing all kinds of queries, running these operations in the background on both Android and iOS.

With the help of sqflite it was possible to create functions to use in screens where there was a need for it, and since these actions are executed in the background, it was mandatory to use asynchronous functions. First, it was inevitable to start by creating the database so that it would be possible to perform the desired operations on it. The database creation followed the form mentioned in the previous chapter and relied on the presented and explained tables. After that, we could then implement the desired actions for the database, in which actions were created to obtain the data of all meals and the meals of a particular day. For symptoms, we followed the same path and obtained a method to obtain all symptoms and another to obtain the symptoms of a particular day. For the Photo, a method had to be created to get the photo for the chosen day and get all the photos in the database.

After the methods for obtaining information from the database were ready, it was essential to have methods that saved the information in the proper table, using for this the insert query. The methods created for the three tables share the same ideology: receive as a parameter of the meal, the symptom or the photo you want to insert and use the data of the input variable to fill the corresponding table in the database. Finally, methods were created to be able to eliminate data from the database, being possible to eliminate one element, several or even all the records present in the table.

With all the essential methods created and working as intended, they could then be used so that it was possible to fill the screens with stored information or fill the database with data entered by the user on the corresponding screen. More background on the methods introduced here will be given later, as a concrete context for their use in the application will be given.

5.3 Screens Integration

With the essential screens already created, it is necessary to form a connection between them, using the architecture discussed above, whenever pertinent. Some screens do not need to access information other than that present on the screen, eliminating the need to connect the widgets to the Store. For example, some of the first screens of the application, such as the Cover and the Registration page, only need the buttons therein to have the *OnClick* function adequately configured for the user to be directed to the corresponding page. However, on the Registration page, it is still necessary to configure the *OnClick* function so that it requests the server to be able to register the user.

Moving on to the pages that need access to the Store, we start with the Login page, which in a certain way resembles the Registration page since it also needs to communicate with the server to verify the user's data when the button is pressed. However, unlike the latter, the Login page also needs to access the Store, which is made available by the *StoreProvider* and can be accessed inside the *StoreConnector*, to be able to, in case the login is verified, save the user's id and retrieve that user's data from the server. This saved id will be helpful on the start page since, after the login, it is necessary to collect the information about the user's meals, symptoms and photos previously stored on the server's database. This information from the server is stored in the local database in order to allow faster access. In the construction of the page, the local database is accessed using the method of obtaining the meals for a specific day, this being, initially, the current day, saving that information in the *refList* variable in the Store. With this information, it is possible, as seen previously, to fill the page with the existing meals, and the user can change the day of visualisation whenever the user wants, which brings up the method previously mentioned, using the day chosen by the user. Transitioning to the Symptoms page, this has a behaviour similar to the previously mentioned page in that, in its construction, the local database is accessed using the method of obtaining symptoms for a specific day, in which the current day is initially used, saving that information in the Store's variable *sintList*. As seen in the previous chapter and using the information stored in the Store, it is possible to fill the page with the symptoms the user has already entered. As for the Daily Graphic screen, what happens is similar to the two previously mentioned screens, in which case it is necessary to access the database to get the photo of the current day or the day chosen by the user.

Regarding the screens that are still to be mentioned, these are where the use of the Store is more important since it will be necessary to obtain information from several screens and widgets to be able to register in the database. First, we have a screen that allows the user to enter the data related to a meal to store in the database. As we can see in Fig. 4.9, the user starts by selecting

the name of the meal from a list in a *Dropdown*. This widget is not built into this screen, being necessary to register the value that the user chose in order to be accessible in the main add meal screen. This registration is done in the State provided by the *StoreProvider*, using the Action intended for this, as previously discussed. After that, it is necessary to store the selected food in the State through a *Dropdown* that, like the previous value, is built outside the main screen, which makes it necessary to store the value so that it can be accessed and processed outside that screen. After collecting all the information that cannot be obtained on this screen, and once the user presses the button to add the meal, the information inserted on the screen is joined with the data saved in the Store. With this, it is possible to register that meal in the database. The last saved meal is saved in the variable present in the State, so, besides using the method to insert the meal in the database, it is also necessary to use the Action that allows changing the State. The add symptoms screen works very similarly to the previous one, with the difference that the user only needs to save the information of one widget that is not built in the referred screen. After the user enters the desired information and presses the button at the bottom of the screen, the method that allows saving the information about the symptom entered in the database is called, also using the Action so that the last symptom stored is always saved in the State.

Finally, the "Profile" screen allows the user to edit their profile and log out of their account. Regarding profile editing, this is not configured since it was necessary to connect to the server to edit the information there. That connection was not complete, as will be discussed below. However, it is possible to log out through the " Logout " button. What happens when that button is pressed is to use the Action that allows erasing the user id stored in State, and after that, the user is redirected to the login page.

5.4 Server Structure

The server, as mentioned before, was built in *NodeJs* using *Koa*, however it was not possible to have the server completely configured due to lack of time. Despite this, the complete configuration of the server will be discussed. Initially, it was necessary to install *NodeJs* in the machine that would host the server in order to be able to configure the server. After that, we went on to the configuration itself, starting by inserting the *Koa* module in the server and then by attributing a port to allow communication with the server.

With the existence of this table, it was considered pivotal to add the corresponding user Id in the Meal and Symptoms tables. After the database was created, it was essential to develop functions that would make it easier for the server to interact with it in order to meet the requests made. Next, the necessary routes and actions to be performed by the server were created in order to respond, whenever possible, adequately to the request made. For this, routes are needed for registering a new user, checking login credentials, logging a meal, registering a symptom, searching for meals and fetching for symptoms, the latter two being relative to one user. With the routes created, the server can answer the request made by using the functions that help interact with the database. It is intended that after configuring the server, the structure obtained is the one shown

Operation	Request	URL (example)	Input Parameters	Response
User tries to login	GET	https://server_name/login/:password_key	password_key - hash created with the password inserted by the user	A JSON with UserId if Login was successfull Result_code = 400 if User does not exist Result_code = 404 if Password is incorrect
Obtain meals from User	GET	https://server_name/getUserMeals/:userId	userId - User Identifier	A JSON with all the User's Meals Empty if User does not have registered Meals
Obtain symptoms from User	GET	https://server_name/getUserSymp/:userId	userId - User Identifier	A JSON with all the User's Symptoms Empty if User does not have registered Symptoms
Register User	POST	https://server_name/addUser	Payload - A JSON with User information	Result_code = 200 if register was a success Result_code = 400 if User already exists
Add Meals to Server Database	POST	https://server_name/addMeals	Payload - A JSON with Meals information	Result_code = 200 if Meals were stored correctly If an error occurs, the Result_code is equal to the MealId that caused the error
Add Symptoms to Server Database	POST	https://server_name/addSymptoms	Payload - A JSON with Symptoms information	Result_code = 200 if Symptoms were stored correctly If an error occurs, the Result_code is equal to the SymptomId that caused the error
Delete Meal from Database	DELETE	https://server_name/deleteMeal/:mealId	mealId - Meal Identifier	Result_code = 200 if Meal was deleted Result_code = 400 if an error occurred
Delete Symptom from Database	DELETE	https://server_name/deleteSymptom/:symptomId	symptomId - Symptom Identifier	Result_code = 200 if Symptom was deleted Result_code = 400 if an error occurred

Table 5.1: REST API

in Fig. 5.3, with the server receiving requests from the application and forming the response by sending it as a *JSON*.The requests made to the server are organized into different categories in accordance with the action to be performed on the server, as can be demonstrated by Fig. 5.4 .

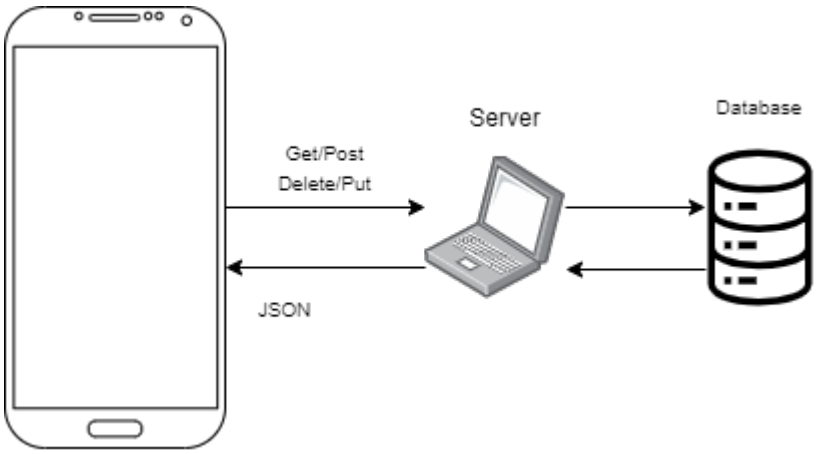


Figure 5.3: Server API

Get	Requests made to obtain information from the server.
Post	Requests made to store the information sent with these.
Delete	Requests made to delete information stored.
Put	Requests made to update information on an existing item.

Figure 5.4: API

The communications that would be expected to have with the server are those possible to visualise in Tab. 5.1, being possible to identify the several requests that are possible to make, as well as the expected answers from the server.

As previously mentioned, the server is not complete, and it is not possible to register users, meals or symptoms, therefore, it is not possible to obtain data from them. For this reason, the

desired authentication mentioned in a previous chapter was not implemented, but it can be guaranteed that the password will not be saved directly in the database since the user does not send it.

Finally, the server provides a response according to the request and what the database interaction function was able to find out, and this response may be the result of the function, a success message or an error message.

5.5 Server Integration with App

Once the server was created, it was still necessary to connect the application to it in order to be able to make the desired requests. Initially, a new user arriving at the application needs to register to be able to access the application and for all their data to be stored on the server. For this, the user must access the registration screen, enter the data correctly and press the registration button there, which, after checking the fields, sends the data to the server to complete the registration. After the user is registered, he must log in, accessing the Login page where he introduces his data. Once the Login button is pressed and following the authentication process presented above, the application sends this data to the server to verify this user's existence and confirm his password. When the authentication data is verified, the server sends the application a successful response, thus allowing the user to proceed to the home screen.

In case the login is successful, it is intended that the application makes requests to the server in the background to get the logged-in user's meals, symptoms and photos data. This data from the server is stored in the local database in order to speed up the requests made by the screen.

There is also the need to store on the server all the meal, symptom, and photo records that have not yet been stored, so it is necessary to send them to the server. As a solution, it was decided to send all the records not yet stored on the server also in the background as soon as the user chooses to log out. After the user presses the logout button, the background process is started, in which several requests will be made to the server to send all the data from the database that is not there yet. It is possible to verify which data are not stored in the server by using the *IsStored* field existing in all the local database tables, which is destined to get the value True if that meal, symptom or photo already exists in the server, thus sending all of them in which this field is set to False. Sending the data only after logging out was the best way found to synchronise the information on the server with the one on the local database so that there are no duplicate values on both sides.

Chapter 6

Conclusions and Future Work

6.1 Overview

This project's main goal was to create a mobile application that would provide patients diagnosed with hypoglycemic events after undergoing bariatric surgery with a tool to assist in more accurate diagnosis and monitoring by the health professional.

With this goal in mind, the final product of this project emerged: the application allows the user to register each meal made by him throughout the day. Each meal, besides the name, date and time, can contain a photo that illustrates the meal itself, as well as a description of what was ingested, and there is also the possibility to indicate each food and its quantity in detail, as well as record some treatment performed. With detailed information about each meal, the user can better understand his body's reaction to a particular food and compare various body reactions with various meals.

The application also allows users to register each symptom they experienced throughout the day. Each symptom, besides the date and time it started, also contains a description to be filled in, allowing the user to choose several symptoms from a list that contains the most common ones. Once the difficulty arose in connecting to the sensor that would allow the values to be obtained, we opted to have a screen where the user could insert a photo of the daily graphic generated by the LibreLink application. With this, the user can see the impact of the meal on his blood glucose, thus improving the diagnosis.

The application is incomplete, as it lacks some essential aspects for its good functioning, the server and the web page. As far as the server is concerned, this was with the database implemented in its entirety, also counting the necessary actions so that changes could be made. It remained to be configured the understanding of the various requests that could be made, as well as the construction of the answers that would be returned to the application. As for the web page referred to, no development was made on it, so it is still at the starting point. The reason for not being concluded was the time spent connecting the screens to the database since this proved to be a somewhat complicated process.

To summarise, what was completed at the time of writing was the mobile application for recording meals and symptoms. The server was left with an initial configuration containing the database responsible for storing all the application information. However, the server could not receive requests since the reception and response configuration was not finished, thus preventing communication with the application.

6.2 Future Work

The application is in a state where it can be used but would not be viable and safe to use, being able to only make records in the local database without these being synchronised on the server. This is not possible since the server is not configured for that, not giving the user the best experience. This is an aspect to be improved in the future as the server is an essential part of the application as it allows the user to have the data stored elsewhere, giving the health professional the possibility to access it to help in the treatment.

Another feature not present in the project is the web page allowing the health professional to access the information their users enter in the application. With this, it was intended that the user who uses the application does not have to go to the doctor to get help from him if necessary. This improvement would be an added value for the health professional since he would be able to have the information of several users in a single platform, making comparisons, which would help in a better understanding of this condition.

The initial idea would be to have in the application a connection to the sensor that measures blood glucose, and since this was not achieved, it would be essential to try to find a better solution to this problem. By obtaining the values for the application, the user could better understand his body's behaviour, taking into account what was ingested. This would also help the doctor in the diagnosis because it would allow a faster and more intuitive analysis of the blood glucose values with the symptoms and the meals inserted.

Finally, another improvement to consider would be the application's appearance as it may not be considered the most appealing to all users, being this an aesthetic improvement only. This consists of changing the button layout, the theme of the application and improving its flow by making it more fluid.

To conclude the above, for the future, it is committed to doing the complete configuration of the server so that it will be possible to communicate with it, allowing the user to have his data available wherever he logs on. Also for the future is the development of the Web platform that aims to give the health professional the ability to access the data that each user entered in the application, thus allowing a better monitoring of the patient and a better diagnosis from these data.

References

[1] C. B. Lobato, M. Guimarães, and M. P. Monteiro. Hipoglicemia apÓs cirurgia bariÁtrica: Da evidÊncia cientÍfica À prÁtica clÍnica. *Revista Portuguesa de Cirurgia*, 50:33–42, 8 2021. doi:10.34635/rpc.884.

[2] Carolina B. Lobato, Sofia S. Pereira, Marta Guimarães, Tiago Morais, Pedro Oliveira, Jorge P. M. de Carvalho, Mário Nora, and Mariana P. Monteiro. Use of flash glucose monitoring for post-bariatric hypoglycaemia diagnosis and management. *Scientific Reports*, 10:11061, 7 2020. doi:10.1038/s41598-020-68029-8.

[3] Abbott Diabetes Care. *FREESTYLE LIBRE 2 FLASH GLUCOSE MONITORING SYSTEM: User Manual*. Abbott Diabetes Care, 12 2020. Last access on 02/2022. URL: https://freestyleserver.com/Distribution/fxaa20.aspx?product=ifu_art41007_114&version=latest&os=all&region=ous&language=xx_yy.

[4] FreeStyle Libre. Sensor freestyle libre 2. https://www.freestylelibre.pt/catalog/product/view/id/561/. Last access on 02/2022.

[5] NightscoutFoundation. xdrip. https://github.com/NightscoutFoundation/xDrip. Last access on 02/2022.

[6] K. Braune, M. Wäldchen, K. Raile, S. Hahn, T. Ubben, S. Römer, D. Hoeber, N. J. Reibel, M. Launspach, O. Blankenstein, and C. Bührer. Open-source technology for real-time continuous glucose monitoring in the neonatal intensive care unit: Case study in a neonate with transient congenital hyperinsulinism. *J Med Internet Res*, 22:e21770, 12 2022. doi:10.2196/21770.

[7] B. Kovatchev. Glycemic variability: Risk factors, assessment, and control. *J Diabetes Sci Technol*, 13:627–635, 6 2019. doi:10.1177/1932296819826111.

[8] Ghalib Ahmed Tahir and Chu Kiong Loo. A comprehensive survey of image-based food recognition and volume estimation methods for dietary assessment. *Healthcare*, 9(12), 2021. URL: https://www.mdpi.com/2227-9032/9/12/1676, doi:10.3390/healthcare9121676.

[9] Inc Figma. Figma. https://www.figma.com. Last access on 07/2022.

[10] Adobe Systems. Adobe xd. https://www.adobe.com/pt/products/xd.html. Last access on 07/2022.

[11] Inc Figma. Figma. https://www.figma.com/figma-vs-adobe-xd/. Last access on 07/2022.

[12] Google. Flutter. https://flutter.dev. Last access on 07/2022.

[13] Inc. Meta Platforms. React native. https://reactnative.dev. Last access on 07/2022.

[14] Soe Thandar Aung, Nobuo Funabiki, Lynn Htet Aung, Hein Htet, Htoo Htoo Sandi Kyaw, and Shinji Sugawara. An implementation of java programming learning assistant system platform using node.js. In *2022 10th International Conference on Information and Education Technology (ICIET)*, pages 47–52, 2022. doi:10.1109/ICIET55102.2022.9779047.

[15] Koajs. https://koajs.com. Last access on 07/2022.

[16] OpenJS Foundation. Openjs foundation. https://openjsf.org. Last access on 07/2022.

[17] Google. Appbar class. https://api.flutter.dev/flutter/material/AppBar-class.html. Last access on 07/2022.

[18] flutter.dev. image_picker: 0.8.5+3. https://pub.dev/packages/image_picker. Last access on 07/2022.

[19] flutter.dev. flutter_redux: 0.10.0. https://pub.dev/packages/flutter_redux. Last access on 07/2022.