Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores

Nutr.io - Multi-platform application for diabetics' nutritional choices

Final release

Authors:

Pedro Pires	**Miguel Luís**	**David Albuquerque**
42206	43504	43566
A42206@alunos.isel.pt	A43504@alunos.isel.pt	A43566@alunos.isel.pt

Tutor:
Fernando Miguel Gamboa de Carvalho
mcarvalho@cc.isel.pt

September, 2020

Instituto Superior de Engenharia de Lisboa

Nutr.io - Multi-platform application for diabetics' nutritional choices

42206 - Pedro Miguel Sequeira Pires

Signature: _____

43504 - Miguel Filipe Paiva Luís

Signature: _____

43566 - David Alexandre Sousa Gomes Albuquerque

Signature: _____

Tutor: Fernando Miguel Gamboa de Carvalho

Signature: _____

Abstract

The idea that every field of study can be digitalized in order to ease monotonous tasks is continuously growing in the modern world. Our project aims to tackle the field of Type 1 diabetes, given its growing prevalence in the world.

One of those monotonous tasks is the count and measurement of carbohydrates in meals used to administer the correspondent amount of insulin, along with their blood levels, to maintain a healthy lifestyle. A task that heavily relies on having access to food databases and realize of how many portions a meal has - usually by using a digital balance or doing estimations.

Eating in restaurants is the perfect example that showcases a gap in this field, that our project, Nutr.io, aims to fill. Most nutritional applications do not provide data for restaurants' meals, such as MyFitnessPal, nor does the user bring his digital balance from home - resulting in a faulty carbohydrate count and therefore the administration of an incorrect insulin dose.

The main goal of this project is to design a system that offers a way to facilitate difficult carbohydrate measurement situations, like in restaurants. To that end, a system that stores meals' nutritional information will be developed, where users can use and calibrate its data with their feedback.

This system will offer an Android application and a front end web application where users can search for nearby restaurants and their respective meals and ingredients. By signing up, the user will be allowed to build insulin profiles which, alongside with nutritional information provided by meals and ingredients, can calculate and provide an accurate insulin dosage that is unique to each user and its medical profile.

Glossary

- **HTTP** - Hypertext Transfer Protocol: An application protocol for distributed, collaborative, hypermedia information systems (RFC 7540);
- **API (Aplication Programming Interface)**: A computing interface which defines interactions between multiple software intermediaries;
- **Framework**: An abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific software;
- **Relational database**: A digital database that provides a mechanism for storage and retrieval of information based on a relational model of data;
- **No-SQL database**: A digital database that provides a mechanism for storage and retrieval of information without a relational model of data;
- **Single-page application**: A web application or website that interacts with the web browser by dynamically rewriting the current web page with new data from the web server;
- **Multi-page application**: A web application or website that interacts with the web browser where pages are rerendered every time there is a data change or submission;
- **DTO (Data Transfer Object)**: An object that carries data between processes;

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Objectives . . . 1
- 1.3 Report structure . . . 1

2 Components and requirements **3**
- 2.1 Requirements analysis . . . 3
  - 2.1.1 Database . . . 3
  - 2.1.2 HTTP Server . . . 3
  - 2.1.3 Mobile application . . . 4
  - 2.1.4 Front web application . . . 4
- 2.2 Component's structure . . . 4
- 2.3 Stakeholders . . . 5
  - 2.3.1 User . . . 5
  - 2.3.2 External API . . . 5

3 Project development **7**
- 3.1 Issues . . . 7
  - 3.1.1 Relational database . . . 7
  - 3.1.2 Food API's . . . 7
  - 3.1.3 Android client . . . 8
  - 3.1.4 Restaurant APIs . . . 8

4 Results **9**
- 4.1 Relational database . . . 10
  - 4.1.1 Used tecnologies . . . 10
  - 4.1.2 Conceptual model . . . 10
- 4.2 HTTP server . . . 11
  - 4.2.1 Used tecnologies . . . 11
  - 4.2.2 Code structure . . . 12
  - 4.2.3 JDBI . . . 12
  - 4.2.4 Spring Security . . . 13
- 4.3 Geolocation . . . 15
- 4.4 Android application . . . 16
  - 4.4.1 Used tecnologies . . . 16
  - 4.4.2 Code structure . . . 16
  - 4.4.3 Local data storing . . . 17
  - 4.4.4 User authentication and authorization . . . 18
  - 4.4.5 Data synchronization . . . 18
  - 4.4.6 Android version compatibility . . . 19

4.4.7 Functionalities 19
4.5 Web browser application 19
4.5.1 Used tecnologies 19
4.5.2 Code structure 20
4.5.3 Functionalities 20

5 Conclusion **21**
5.1 Future development 21

6 Appendices **23**

List of Figures

2.1 Nutr.io platform components 5

4.1 Database conceptual model 10
4.2 Server's classes structure 12
4.3 JDBI using a fluent API style 13
4.4 JDBI using a SQL object API style (example from HTTP server) 13
4.5 The JWT workflow 14
4.6 A Spring security workflow example with the POST /user/login 14
4.7 The repository pattern diagram 17

1 Initial plan accorded in the project's proposal 24
2 API nutritional accuracy sheet 29

Chapter 1

Introduction

1.1 Context

TODO - Pedro

1.2 Objectives

- Design a system that helps individuals with type 1 diabetes easing difficult carbohydrate measurement situations, specifically in restaurants.
- Build a platform maintained by its community, using users' submissions to improve the data's accuracy;
- Deliver a mobile application where the user can search nearby restaurants and their meals;
- Design an insulin calculator that computes insulin dosages based on the nutritional information of the meals selected by the user;
- Protect user's sensitive data, such as insulin profiles, via encryption.

1.3 Report structure

This report states every issue encountered during project's development, mentioning the decisions the group made to solve them. This might also include changes in the initial plan, that the group found relevant for the project's progress efficiency.

The diagrams and schemas developed for this project are shown when approaching the respective topic, however there is an appendix which contains additional information about the project, having references pointing to it when necessary.

Chapter 2

Components and requirements

2.1 Requirements analysis

In order to build this multiplaform application a relational database and a HTTP server must be included in the backend, which will store and supply information to the mobile and web browser clients.

The next subsections identify the requirements for each component.

2.1.1 Database

Functional requirements

- A model that stores and organizes information from diferent sources - APIs and users;
- Unify various types of information from various sources into a single entity, named Submission;
- Suggest meals for a given restaurant;
- Store user sensitive information;
- Provide restaurants around any given geolocation;
- Seperate data, labeling which ones should be votable, favorable or reportable;

Non-functional requirements

- Performance when searching nearby restaurants;
- Query simplicity;
- Database normalization;

2.1.2 HTTP Server

Functional requirements

- Provide endpoints that allow users to view and create nearby restaurants and meals;
- Provide users the ability to edit their submissions and vote or report others;
- Filter content based on a submission's votes and reports;

Non-functional requirements

- Garantee authentication and password encryption when registering;
- Encrypt authenticated users' insulin profiles when inserting them into the database;

2.1.3 Mobile application

Functional requirements

- Communicate with the HTTP server in order to display nearby restaurants and their meals based on current geolocation;
- Allow users to create restaurants and meals;
- Allow authenticated users to create and remove insulin profiles;
- Calculate insulin dosages based on a user's blood glucose, insulin profile and the selected meal;
- Allow users to vote, report and add submissions to their personal favorites;
- Allow mimimum functionalities for unauthenticated users;

Non-functional requirements

- Allow the user to choose its default measurement units;
- Allow user authentication and registering;

2.1.4 Front web application

Functional requirements

- Platform administration tools;
- Faulty data management;
- User control;

Non-functional requirements

- Responsive and performant UI;

2.2 Component's structure

To match previously stated requirements the group conceived a platform following the structure represented by the next picture:

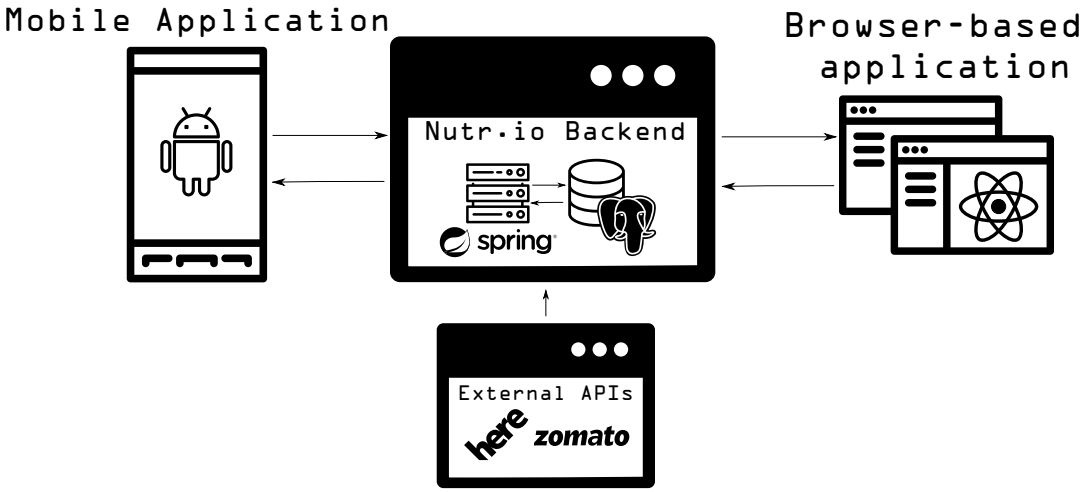


Figure 2.1: Nutr.io platform components

As shown, the platform will be composed by two clients: a mobile application for Android devices and a web browser application, which will have as backend a HTTP server and a relational database.

External APIs will also be used to obtain restaurants' related information. After some discussion, the group chose to use the Here API, to provide restaurant information and geolocation.

More details about the chosen tecnologies for each component will be described in the fourth chapter of this report.

2.3 Stakeholders

Analyzed components and requirements imply the existance of two types of submitters which will interact with the system - an user and external APIs.

The way they can interact with the system is as follows:

2.3.1 User

An user interacts with the system via a client which can request submissions. Additionally, if authenticated, an user can create, vote and report submissions; making him the most vital stakeholder when it comes to creating a self-maintained system.

2.3.2 External API

An external API interacts directly with the HTTP server and is resposible in providing nearby restaurants around given geolocation.

However, since not all restaurants are registered in utilized APIs, the need for a collaborative user is highly vital when creating a self-maintained system.

Chapter 3

Project development

3.1 Issues

This section describes the issues found and the decisions made to overcome them during development.

3.1.1 Relational database

Multiple iterations of the database's model were designed until a consistent result was implemented, mainly due to a lack of initial research on the project's functional requirements and a later change of some core beliefs (ideas?) regarding how meals would be handled. (maybe reference another problem section?)

3.1.2 Food API's

While testing an initial version of the HTTP server, it was concluded that no researched API responsible in providing accurate nutritional information exists.

This conclusion came after comparing the nutritional values of multiple meals and ingredients from three APIs - Edamam, Nutrionix and Spoonacular - to the corresponding values provided by official portuguese sources. The results can be found in TODO

The group assumes that this inaccuracy is due to the fact that mentioned API's automatically calculate a recipe's carbohydrates from its ingredients without taking into consideration the cooking process, meaning that, for example, 100g of raw rice does not equal to the same amount of carbohydrates that 100g of cooked rice might have.

Another possible explanation is that there is no international standard for nutritional values, meaning that the same meal (and it's ingredients nutritional composition) can have different carbohydrates between Portugal and the United States of America.

As a result, the system no longer depends on food APIs meaning that every meal, ingredient and its' nutritional values need to be inserted manually by other sources, such as the developers themselves or authenticated users. For an initial release, the group added around 60 basic ingredients and 30 meals.

This comes with the limitation that certain ingredients might have been missed, meaning that a user might not be able to create their desired meals.

3.1.3 Android client

The Android application's development progressed normally but it had to be put on hold sometimes, because of HTTP server's endpoints' completion, in which the application depends strongly. A major dto and model restructure had also to be made inside the mobile application in order to meet with the current HTTP responses.

3.1.4 Restaurant APIs

During early stages of development, the group incorrectly assumed that researched restaurant APIs (namely Zomato) were capable of providing menus for any restaurant in text form, and as such, their meals. Although such endpoint exists [Reference here], restaurant owners instead prefer to publish their menu as an image, meaning that initially established functional requirements would never be able to be accomplished without another viable alternative.

As such, two alternatives were discussed:

An OCR tool could be used in order to convert meals from a graphical menu to a text-based one. This implies additional response times for the user when querying a restaurant due to the added layer of communications [esta parte podia ser melhorada] and is not reliable in cases where restaurants design their menus with just an image of a meal and its' price [example reference/image].

Another alternative - and the adopted one - is to manually establish a set of statistically probable meals for cuisines and suggest them when obtaining a restaurant's information and cuisines. This solution is based on statistics and cultural assumptions meaning that not every suggestion might be accurate, which is why authenticated users' input is highly valuable in maintainin the system's data.

Chapter 4

Results

This chapter highlights the final results of the project, including every decision made by the group and tangible accomplishments for each developed module.

4.1 Relational database

Which database should be used is one of the first and most crucial steps after determining functional requirements as it establishes the data's model and how the communication with the server will be made, meaning that a wrong choice would cause major restructures.

Before adopting a specific database, one should first consider between a relational model and a No-SQL model. Given the complex hierarchy between data entities, tutor's considerations and the need for a model capable of providing quick results when querying around a geolocation, a relational model was chosen.

4.1.1 Used tecnologies

When the project was in a planning phase, the group decided that a relational database was the most suitable option for this project instead of a No-SQL database, because of the project's structure - there are many hierarchies between entities, which invalidated the no-SQL option.

Out of all researched relational databases, PostgreSQL was chosen as it supports the PostGIS plugin, and is supported by Heroku - allowing to deploy the application in later stages of development.

4.1.2 Conceptual model

As a result of multiples redesigns, here is the database's conceptual model.

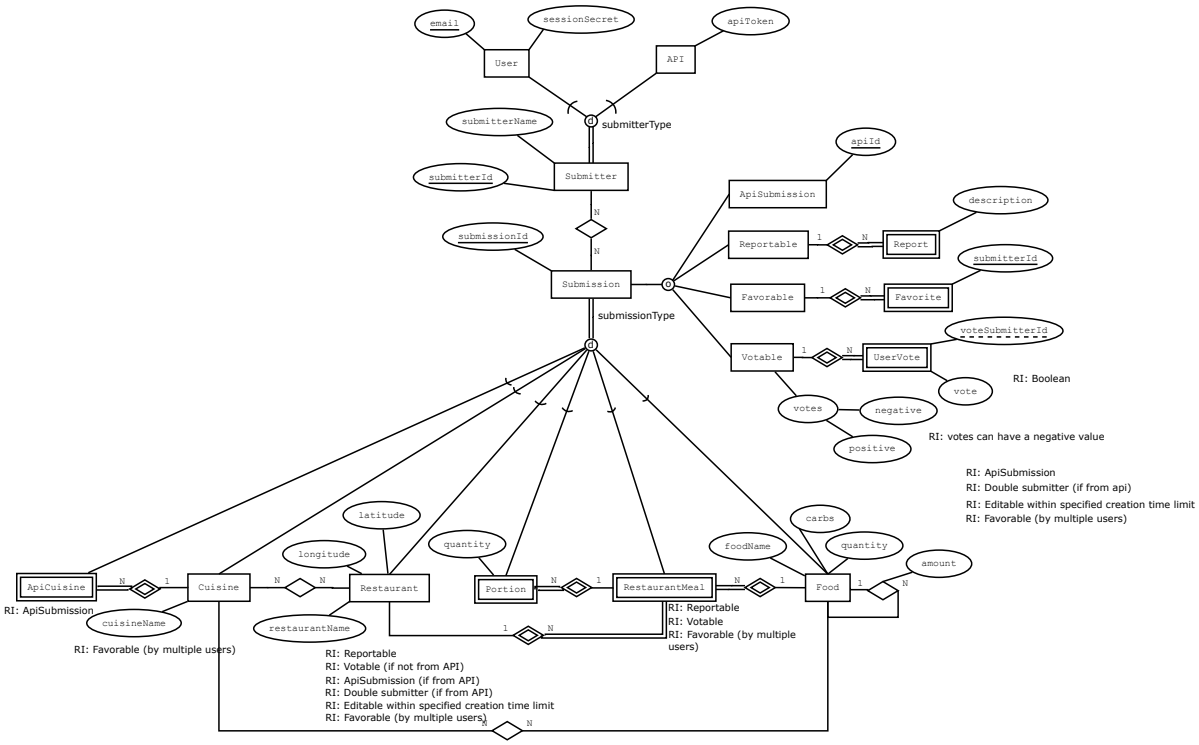


Figure 4.1: Database conceptual model

The database's relational model is present inside this report's appendix [Appendix B - Database

relational model].

In the relational model there are tables which are not specified in the conceptual model. These are a product from associations between entities which will simplify queries' complexity.

Now the submission can fall into 4 categories: ApiSubmission, Reportable, Favorable and Votable, in order to disguish between submissions that are from the user or from APIs and to separate which ones can be reportable, favorable and votable by the user.

The cuisine entity has now an associated entity called ApiCuisine, to save cuisine information provided by the Here API.

Meals and ingredients were now condensed into one entity called food - now each meal can have meals inside it that can also be considered ingredients in other contexts.

Therefore each meal possesses nutritional information, which is essential to the user especially to the insulin calculations. That information is composed by 'carbs' - meal's carbohydrates; and quantity - meal's quantity.

4.2 HTTP server

4.2.1 Used tecnologies

Kotlin

The group chose to use Kotlin[12] for the HTTP server developed as it is a language that is being more adopted and used nowadays and because it is totally interoperable with Java[5].

It was also the language used during PDM, which is an optional course for Android application development inside the LEIC programme, making this a language the group felt confortable with.

Spring MVC

At the beginning of the project the group decided to use Spring MVC[17] rather than Ktor[13], as the first one is taught in DAW, which is an optional course for Web applications development inside the LEIC programme. As Spring MVC has a better coverage inside the LEIC programme, the group considered it a more solid choice.

Used dependencies

Here are all the dependencies injected inside HTTP server gradle settings file.

- **Kotlin base dependencies** - kotlin-reflect and kotlin-stdlib-jdk8;
- **Spring base dependencies** - spring-boot-starter and starter-web;

- **Mockito** - for tests with mocks;
- **Jackson** - for JSON serialization and deserialization;
- **JDBI** - the driver/interface for connecting with the relational database;
- **Spring Security** - for authentication and authorization proposes.

4.2.2 Code structure

TODO - eng. Félix says: "Needs more information about the backend organization, such as: intermediaries, controllers, services, database access method, external dependencies used"

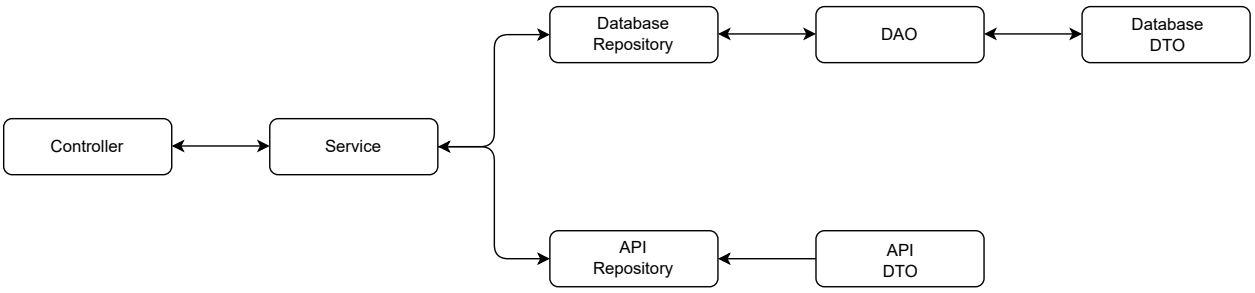


Figure 4.2: Server's classes structure

4.2.3 JDBI

After some discussion of which driver should be used to allow communication between the server and database, the group decided that the JDBI[8] was the best option as it is a library built on JDBC[7].

TODO - eng. Félix says: "Usage of JDBI and the declarative API needs justification..."

The library also exposes two different API's styles: a fluent style and sql object style (used during development), as shown below.

```java
// using in-memory H2 database
DataSource ds = JdbcConnectionPool.create("jdbc:h2:mem:test",
                                          "username",
                                          "password");
DBI dbi = new DBI(ds);
Handle h = dbi.open();
h.execute("create table something (id int primary key, name varchar(100))");

h.execute("insert into something (id, name) values (?, ?)", 1, "Brian");

String name = h.createQuery("select name from something where id = :id")
                    .bind("id", 1)
                    .map(StringColumnMapper.INSTANCE)
                    .first();

assertThat(name, equalTo("Brian"));

h.close();
```

Figure 4.3: JDBI using a fluent API style

```kotlin
@SqlQuery( value: "SELECT * FROM $table WHERE " +
        "ST_Distance(" +
        "ST_MakePoint($table.$latitude, $table.$longitude)::geography," +
        "ST_MakePoint(:latitude, :longitude)::geography, " +
        "false" +
        ") <= :radius"
)
fun getByCoordinates(@Bind latitude: Float, @Bind longitude: Float, @Bind radius: Int): Collection<DbRestaurantDto>

@SqlQuery( value: "SELECT * FROM $table WHERE $id = :submissionId")
fun getBySubmissionId(@Bind submissionId: Int): DbRestaurantDto?
```

Figure 4.4: JDBI using a SQL object API style (example from HTTP server)

4.2.4 Spring Security

JSON Web Tokens

As each client needs to have authentication to provide the user a way to create an account and allow submissions and data synchronization, the group had to discuss about the platform's security and the safest ways to do that.

It was concluded that the use of **JSON Web Tokens**[10] was the best option, because of the nature of the clients, more specifically the fact that the mobile application is completely **stateless**.

Another advantage is the fact that these tokens have an expiration time, which means that after a certain amount of time they are no longer valid. In case of security breach, this feature becomes useful, because if the attacker does not have a way to generate valid tokens neither does not know the user password and steals a valid token, this will only be used by a short period of time (10h), easing the amount of damage that an intruder can make and confining it to only one user inside the platform.

The picture below represents a very generic and simplified workflow of the JSON Web Token.

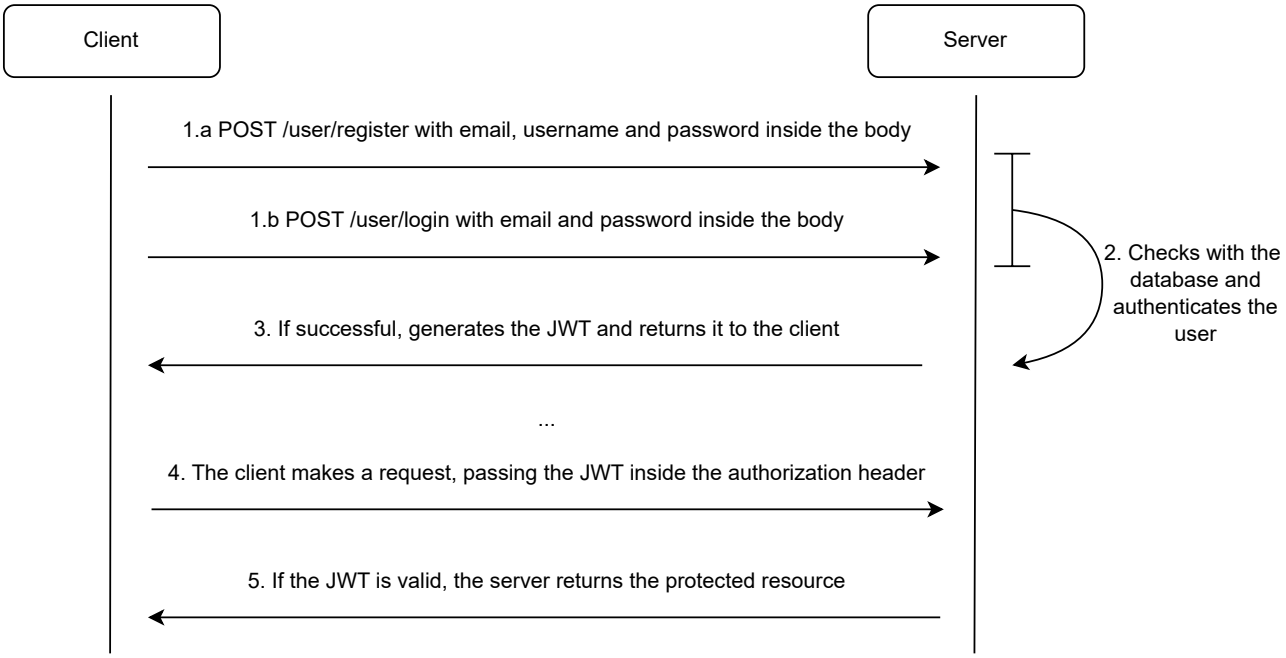


Figure 4.5: The JWT workflow

Implementation

To implement the shown workflow inside the HTTP server, as the group is implementing it with Spring, the more obvious choice was to use **Spring Security**[18].

Spring Security is a customizable authentication and access-control framework.

The picture below shows how the server handles a user login usin this framework.

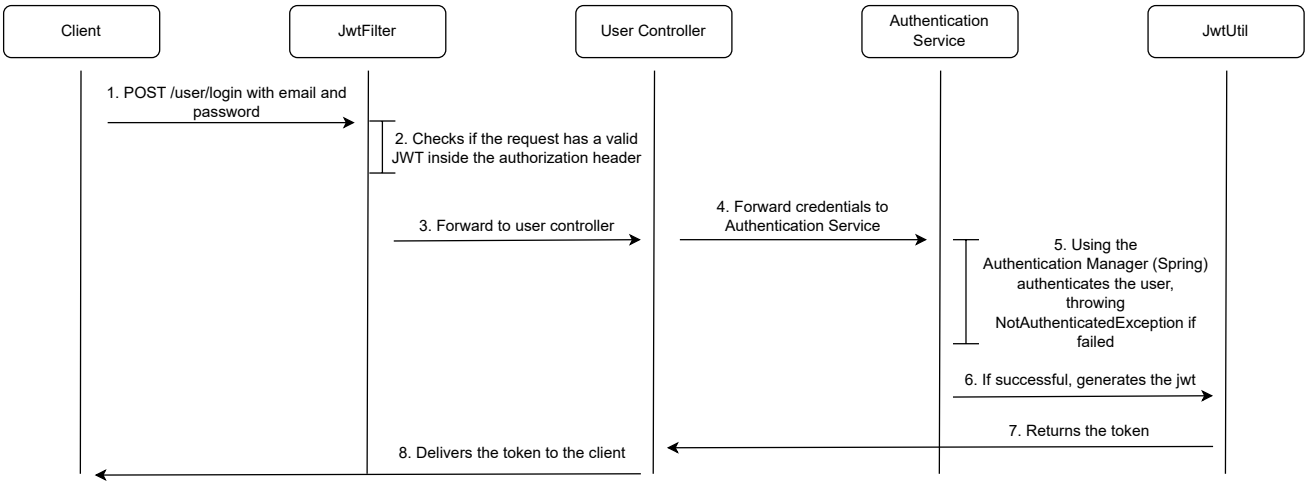


Figure 4.6: A Spring security workflow example with the POST /user/login

After the previously mentioned dependencies are installed, the WebSecurityConfig is the first

class to be constructed. Here are specified, via antMatchers, which endpoints do not need authentication, acting like a whitelist, so every endpoint that is not specified via antMatchers needs authentication, and will return code 401 Unauthorized if the JWT is invalid or absent. The JWT filter is also started up inside this class.

The Jwtfilter class, as the name says, filters each request, checking the authentication header and extracting the jwt from the Bearer verifying if it is valid. This class extends the OncePerRequestFilter() which garantees a single execution of this filter every request.

The Authentication service class calls the Spring Authentication Manager and authenticates the user, it provides methods which call the JwtUtil to retrieve the email from the token or encode the password when registering.

The password encoding always happens when the user registers for the first time: the server hashes the password using **BCrypt** before inserting the new user into the database.

BCrypt[1] is a password-hashing function based on the Blowfish[2] cipher. The group found this function very convenient for these reasons:

- Already pre salts the passwords, preventing rainbow table attacks[14];
- Makes bruteforce attacks inviable: the iteration count can be increased to make it even slower to crack. This cipher makes even GPU-powered bruteforce attacks impracticable due to this feature.

The JwtUtil is the core class which validates, generates and adds claims to the tokens.

4.3 Geolocation

Given how all clients rely on obtaining nearby restaurants, there was a need to implement a geolocation function in the project's design.

Initial research showcased two possible solutions: Haversine distances and cartesian distances, where the latter returns a highly imprecise distances. As such, Haversine was selected.

The next step was to choose which system filters nearby restaurants: database or HTTP server. After some discussion, the group decided that database was the best option for two reasons:

- Given the large amount of existing restaurants, sending such data from the database to the HTTP server so that it could filter it would occupy too much memory;
- PostgreSQL already supplies extensions that add support for location queries, namely PostGIS.

4.4 Android application

4.4.1 Used tecnologies

Kotlin

The group chose to use Kotlin for the mobile application development, as it is now the official programming language for Android development, according to Google.

It is also the language taught during the optinal course - mobile devices programming (PDM)

External dependencies

Here are the dependencies that were included in the mobile application which gave more functionalities to it.

- **Volley** - an HTTP library for Android networking;
- **Jackson** - JSON serialization, deserialization and handling;
- **Room** - A framework to store data locally;
- **MapBox** - A framework to provide maps and geolocation tools;
- **MPAndroidChart** - provides custom graphs inside the application;
- **Glide** - a framework for image loading;
- **Androidx crypto** - a new crypto library made by Google, used to encrypt User credentials.

4.4.2 Code structure

Drawing pattern

The mobile application code structure follows the **repository pattern**, which is a code architecture recommended by the **Android Jetpack**[9] for this type of applications.

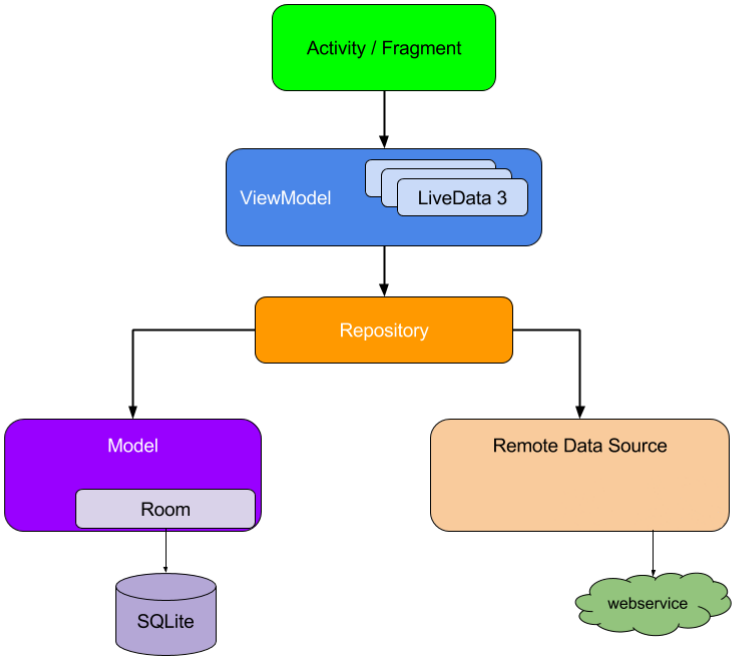


Figure 4.7: The repository pattern diagram

Above there is the pattern's diagram provided by the Android Jetpack.

The idea behind this architecture is that each Activity or fragment has its own ViewModel and each one calls the needed functions present inside the repository. The repository is a layer that manages where the information should be retrieved from.

The 'DTO to model' mapping also occurs inside the repository, following the rule where ViewModels should only manipulate models and the layers below should only use DTOs.

By following this pattern, the code becomes segmented and organized, allowing a good comprehension and code maintainability.

Fragments

The group chose to use fragments[4] for each application view instead of activities. Although a fragment has a more complex lifecycle than the activity and depends on it to exist, they are far more lightweight to instanciate than an activity and thus they provide more performance to the application.

It is also the recommended Android widget to use when designing an application with a side drawer.

4.4.3 Local data storing

As mentioned in the dependencies, the mobile application utilizes Room to store data locally. This is convenient for multiple reasons:

- To allow using the application in offline situations;
- To save data in order to avoid unnecessary requests to the server;
- To help data synchronization, that will be detailed later in this section.

4.4.4 User authentication and authorization

The user has the ability to register and login in the mobile application. Besides being the server responsable for these functions, the mobile application has also some intervention here, because after a successful login or register, the HTTP server will return a jwt (JSON Web Token) that will authenticate and authorize the user in future requests.

This token will be stored in the Android Shared Preferences[16] and it will also to be renewed periodically due to its 10h expiration time. The user credentials will also be saved inside the mobile device to allow automatic logins to renew the user's JSON Web Token and avoid its expiration.

Problem: The content inside the shared preferences is written in plaintext. Is it safe to store user credentials inside the shared preferences?

Although the Android Shared Preferences being a safe place to store application information, this fact is not completely true: a normal device can not access these preferences and it should be a safe place to store user credentials, however rooted devices can easily access the shared preferences file and retrieve plaintext from it, which would compromise the user security.

Resolution: Androidx Crypto

The Androidx crypto[3] was used to solve this issue. This library is used to encrypt the user credentials before writing them inside the mobile device.

These new Google library takes advantage of the Android KeyStore[11] system, which encrypts information using a hardware-level encryption, making the encryption even harder to break. The information is encrypted using a symmetric cipher algorithm (AES-256), the key used to sign and encrypt information is hardware-generated and it is managed by the application itself, so the key's retrieval from an 'encrypted' shared preferences is equal to the 'normal' shared preferences.

The group also discussed if the credentials should be saved inside the device or if only the database should possess them. If that approach was taken, the user had to login each time it was needed to read or write a protected resource.

As this platform is not, for example, a bank application that needs top protection. The group found this level of protection unnecessary for the application and inconvenient for the user and decided that only the essential protection should be provided - user credentials encryption to avoid information leaks from rooted devices.

4.4.5 Data synchronization

Background data synchronization will happen after a successful login or register. The only user data that will be synchronized are:

- Insulin profiles;
- Custom meals made by the user;
- Favorites.

When logged in, the data can be synchronized in two ways:

- the user forces the synchronization by swiping down on a list;
- The Android WorkManager will make sure that the data is synchronized at least once a day when the phone is inactive and connected to the internet.

4.4.6 Android version compatibility

In order to garantee a global support by most of the Android devices nowadays, the mobile application is supported since **Android 7** (API level 24) up to **Android 10** (API level 29).

4.4.7 Functionalities

TODO: application images and diagrams

TODO

As mentioned in the progress report, the group managed to implement in the mobile application a fragment that displays a map with a list of restaurants nearby the user.

The user can also search for restaurants, meals and cuisines providing the associated name or identifier.

The core feature of the application was also finished during this time period - the insulin calculator. This feature calculates how many insulin doses should be injected in order to maintain the blood glucose levels stabilized according to the user's planned glucose objective for that period of the day.

The blood glucose objective is set inside the user's profile, by creating multiples insulin profiles, each one has a limited time period to give the user freedom to map its own insulin routine throughout the day.

This is due to the fact that user's insulin sensitivity factor varies along the day, so the user has the ability to specify its own values in order to the calculator

4.5 Web browser application

4.5.1 Used tecnologies

React framework

The group chose to build the website with JavaScript [6] using the React framework [15] , as it was the framework lectured in the Web applications development course and has innumerous advantages to other frameworks, such as Node.JS.

4.5.2 Code structure

Single-page application

As website design pattern, the group chose to conceive a single-page application. This pattern was chosen for a variety of reasons, being the main one a better performance comparing to a traditional multi-page application.

Routing

Under development

4.5.3 Functionalities

Under development

Chapter 5

Conclusion

This project aimed to fulfil a gap that was present inside almost every other nutrition application - giving nutritional information about restaurant's meals while providing insulin dosages calculations to users with type 1 diabetes.

To this end, two client applications, a HTTP server and a database were developed. The platform's backend provided the clients the most important capability that was accomplished by the end of the project - platform's self-maintainability.

This mentioned characteristic is benevolent when developing a highly scalable platform like this one, where posterior centralized data validations becomes inviable due to large amounts of generated information.

Reached this part of the report, it can be concluded that every functional and non-functional requirement which was initially planned was accomplished by the end of this project. Some considerations about the project's future development were also discussed by the group and are presented in the next section.

5.1 Future development

As this project represents a final bachelor's project, some future work could still be done in order to publish and deploy the platform to the general public, making it more successful and profitable.

Here are some topics the group agreed that could belong to this project's future development:

Statistics analysis

Given that one of this project's main strengths is information storage and mapping, the group found relevant that statistics analysis about submitted restaurants and meals should be done, in order to help and provide other work fields information that could be useful.

However every data that is collected should be properly disclaimed and only the information that would not compromised the user's privacy should be colectable.

Interaction with similar platforms

As written in the last topic, the information could enrich the world of nutrition and health platforms. To allow this, the API could be available to the public, so it could be integrated with other platforms alike.

Improved social system

This platform depends strongly on the community, as such there should be a social system improval in order to make its members more active and the platform's more responsive and fulfilled.

One way to do that could be social rewards for the most active and contributive users.

Client UI/UX revision

As the group that developed this platform is not qualified in this matter, the user interface should be reviewed and improved by an UI/UX team, which has the tools to make the clients' interface more appealing to the users.

Certified members

Anyone can participate and make submissions to the platform. Given this fact, voided information can be submitted and spread inside the community, contributing to misinformation.

To tackle this issue there should be member certifications for the most active members or even certified professionals, which could validate submitted information and make the platform's environment a more trustworthy place online.

Two-factor authentication

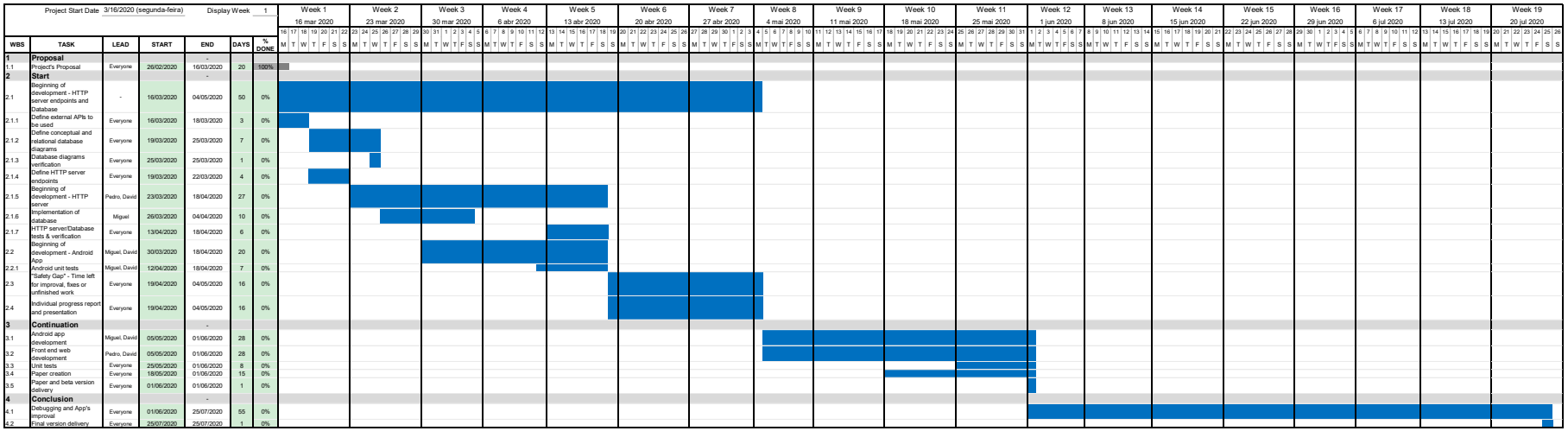
Security is a very important topic nowadays, being a field that is constantly evolving. As it is proven that single-factor authentication might be not enough to avoid attacks completely, a two-factor authentication could be implemented to improve user's security.

Chapter 6

Appendices

This chapter displays all the appendices referenced in this report.

Appendix A - Initial plan



Project Start Date: 3/16/2020 (segunda-feira) — Display Week: 1

WBS	TASK	LEAD	START	END	DAYS	% DONE
1	**Proposal**			-		
1.1	Project's Proposal	Everyone	26/02/2020	16/03/2020	20	100%
2	**Start**			-		
2.1	Beginning of development - HTTP server endpoints and Database	-	16/03/2020	04/05/2020	50	0%
2.1.1	Define external APIs to be used	Everyone	16/03/2020	18/03/2020	3	0%
2.1.2	Define conceptual and relational database diagrams	Everyone	19/03/2020	25/03/2020	7	0%
2.1.3	Database diagrams verification	Everyone	25/03/2020	25/03/2020	1	0%
2.1.4	Define HTTP server endpoints	Everyone	19/03/2020	22/03/2020	4	0%
2.1.5	Beginning of development - HTTP server	Pedro, David	23/03/2020	18/04/2020	27	0%
2.1.6	Implementation of database	Miguel	26/03/2020	04/04/2020	10	0%
2.1.7	HTTP server/Database tests & verification	Everyone	13/04/2020	18/04/2020	6	0%
2.2	Beginning of development - Android App	Miguel, David	30/03/2020	18/04/2020	20	0%
2.2.1	Android unit tests	Miguel, David	12/04/2020	18/04/2020	7	0%
2.3	"Safety Gap" - Time left for improval, fixes or unfinished work	Everyone	19/04/2020	04/05/2020	16	0%
2.4	Individual progress report and presentation	Everyone	19/04/2020	04/05/2020	16	0%
3	**Continuation**			-		
3.1	Android app development	Miguel, David	05/05/2020	01/06/2020	28	0%
3.2	Front end web development	Pedro, David	05/05/2020	01/06/2020	28	0%
3.3	Unit tests	Everyone	25/05/2020	01/06/2020	8	0%
3.4	Paper creation	Everyone	18/05/2020	01/06/2020	15	0%
3.5	Paper and beta version delivery	Everyone	01/06/2020	01/06/2020	1	0%
4	**Conclusion**			-		
4.1	Debugging and App's improval	Everyone	01/06/2020	25/07/2020	55	0%
4.2	Final version delivery	Everyone	25/07/2020	25/07/2020	1	0%

Figure 1: Initial plan accorded in the project's proposal

Appendix B - Database relational model

- **Submitter**
  - Attributes: submitterId, submitterName, submitterType
  - Primary Key(s): submitterId
  - Foreign Key(s): -
  - Not null: submitterName, submitterType
- **User**
  - Attributes: *submitterId*, email, sessionSecret, creationDate
  - Primary Key(s): *submitterId*, email
  - Foreign Key(s): *submitterId* references Submitter(submitterId)
  - Not null: sessionSecret
- **API**
  - Attributes: *submitterId*, apiToken
  - Primary Key(s): *submitterId*
  - Foreign Key(s): *submitterId* references Submitter(submitterId)
  - Not null: apiToken
- **Submission**
  - Attributes: submissionId, submissionType, submissionDate
  - Primary Key(s): submissionId
  - Not null: submissionType
- **ApiSubmission**
  - Attributes: *submissionId*, *apiId*
  - Primary Key(s): *submissionId*, *apiId*
  - Foreign Key(s): *submissionId* references Submission(submissionId)
  - Not null: submissionType
- **SubmissionSubmitter**
  - Attributes: *submissionId*, *submitterId*
  - Primary Key(s): *submissionId*, *submitterId*
  - Foreign Key(s):
    - *submissionId* references Submission(submissionId)
    - *submitterId* references Submitter(submitterId)
  - Not null: *submitterId*
- **SubmissionContract**
  - Attributes: submissionId, submissionContract
  - Primary Key(s): submissionId, submissionContract

- **Report**
  - Attributes: *submissionId*, *submitterId*, description
  - Primary Key(s): *submissionId*, *submitterId*
  - Foreign Key(s):
  - * *submissionId* references Submission(submissionId)
    * submitterId references Submitter(submitterId)
  - Not null: description
- **Votes**
  - Attributes: *submissionId*, positiveCount, negativeCount
  - Primary Key(s): *submissionId*
  - Foreign Key(s): *submissionId* references Submission(submissionId)
  - Not null: submissionType
- **UserVote**
  - Attributes: *submissionId*, voteSubmitterId, vote
  - Primary Key(s): *submissionId*, voteSubmitterId
  - Foreign Key(s):
    * *submissionId* references Submission(submissionId)
    * voteSubmitterId references Submitter(submitterId)
- **Restaurant**
  - Attributes: *submissionId*, restaurantName, latitude, longitude
  - Primary Key(s): *submissionId*
  - Foreign Key(s): *submissionId* references Submission(submissionId)
  - Not null: restaurantName
- **Cuisine**
  - Attributes: submissionId, cuisineName
  - Primary Key(s): cuisineName
- **ApiCuisine**
  - Attributes: *submissionId*, *cuisineSubmissionId*
  - Primary Key(s): cuisineName
  - Foreign Key(s):
    * *submissionId* references Submission(submissionId)
    * *cuisineSubmissionId* references Cuisine(submissionId)
- **Meal**
  - Attributes: *submissionId*, mealName, carbs, quantity, unit
  - Primary Key(s): *submissionId*
  - Foreign Key(s): *submissionId* references Submission(submissionId)
  - Not null: mealName

- **RestaurantMeal**
  - Attributes: submissionId, *restaurantSubmissionId*, *mealSubmissionId*
  - Primary Key(s): submissionId, *restaurantSubmissionId*, *mealSubmissionId*
  - Foreign Key(s):
    - *restaurantSubmissionId* references Restaurant(submissionId)
    - *mealSubmissionId* references Meal(submissionId)
- **Favorite**
  - Attributes: *submissionId*, submitterId
  - Primary Key(s): *submissionId*, submitterId
  - Foreign Key(s): *submissionId* references Submission(submissionId)
- **Portion**
  - Attributes: *submissionId*, restaurantMealSubmissionId, quantity
  - Primary Key(s): *submissionId*
  - Foreign Key(s): *submissionId* references Submission(submissionId)
  - Not null: quantity
- **MealIngredient**
  - Attributes: *restaurantSubmissionId*, *ingredientSubmissionId*, quantity
  - Primary Key(s): *restaurantSubmissionId*, *ingredientSubmissionId*
  - Foreign Key(s):
    - *mealSubmissionId* references Meal(submissionId)
    - *ingredientSubmissionId* references Ingredient(submissionId)
- **RestaurantCuisine**
  - Attributes: *restaurantSubmissionId*, *cuisineName*
  - Primary Key(s): *restaurantSubmissionId*, *cuisineName*
  - Foreign Key(s):
    - *restaurantSubmissionId* references Restaurant(submissionId)
    - *cuisineName* references Cuisine(cuisineName)
- **MealCuisine**
  - Attributes: *mealSubmissionId*, *cuisineName*
  - Primary Key(s): *mealSubmissionId*, *cuisineName*
  - Foreign Key(s):
    - *mealSubmissionId* references Meal(submissionId)
    - *cuisineName* references Cuisine(cuisineName)

Appendix C - Endpoints' table

Method	Path	Query String	Body parameters	Description
GET	\restaurant	float latitude, float longitude, optional String name, optional int radius, optional String name		Search for restaurants and their cuisines, based on location or named search
GET	\restaurant\:restaurantId			Obtain specific restaurant's full information by given restaurantId
GET	\restaurant\:restaurantId\meal			Obtain all suggest and user inserted restaurant meals for given restaurant
GET	\restaurant\:restaurantId\meal\:mealId	int skip, int count		Obtain specific restaurant meal for given restaurantId and mealId
GET	\cuisines	optional int skip, optional int limit		List possible cuisines
GET	\ingredients	optional int skip, optional int limit		Get all possible ingredients
GET	\meal	optional string[] mealTypes, optional int skip, optional int count, optional string[] cuisines		Get all suggested meals
GET	\meal\:mealId			Obtain specific meal's full information by given mealId
POST	\restaurant		RestaurantInput	Create a new restaurant around given geolocation
POST	\meal		MealInput	Create a user meal with at least one ingredient
POST	\restaurant\:restaurantId\meal\:mealId		PortionInput	Insert a new portion for given restaurant meal
PUT	\restaurant\:restaurantId\vote		VoteInput	Add or update your vote on a user restaurant
PUT	\restaurant\:restaurantId\:mealId		VoteInput	Add or update your vote on a restaurant meal created by an user
PUT	\restaurant\:restaurantId\meal		RestaurantMealInput	Creates a restaurant meal from given user meal
DELETE	\restaurant\:restaurantId			Delete user created restaurant
DELETE	\restaurant\:restaurantId\vote			Delete user's vote on an user's restaurant
DELETE	\restaurant\:restaurantId\meal\:mealId			Delete user's portion submission for given restaurant meal
DELETE	\restaurant\:restaurantId\meal\:mealId\portion			Delete user's restaurant's meal portion
DELETE	\restaurant\:restaurantId\meal\:mealId\vote			Delete user's restaurant's meal vote
DELETE	\meal\:mealId			Delete an user created meal, along with any associations with a restaurant the meal might have

Appendix D - API nutritional accuracy sheet

Meal string displays the query String used to search in respective API

Meal (always 100g)	APDP Values	Edamam
	Carbs	Value
Green peas	8	14
Broad bean (favas)	7	11
cooked red kindey beans	14	22
cooked chickpeas	17	27
Soybeans, mature cooked, boiled, without salt	6	9/30
Lupine (tremoço)	7	9
Corn bread	37	43
Wheat bread	57	48
Cooked Rice (simple)	28	28
Tomato Rice	19	18
Roasted Potato (assado)	24	17
potatoes, boiled, cooked in skin, flesh	19	20
potatoes, boiled, cooked in skin, skin	19	17
sweet potato, cooked, boiled, without skin	~17	17
French fries	28	23
Mashed potato	17	16
Pizza	24	29
Chicken rice	25	12
Baked Fish and Rice	15	8
Octopus rice	10	no result

Figure 2: API nutritional accuracy sheet

Bibliography

[1] Bcrypt. URL https://auth0.com/blog/hashing-in-action-understanding-bcrypt/.

[2] Blowfish. URL https://en.wikipedia.org/wiki/Blowfish_(cipher).

[3] Android crypto. URL https://developer.android.com/reference/androidx/security/crypto/package-summary.

[4] Android fragment. URL https://developer.android.com/guide/components/fragments.

[5] Java, . URL https://java.com/en/download/faq/whatis_java.xml.

[6] Javascript, . URL https://developer.mozilla.org/pt-PT/docs/Web/JavaScript.

[7] Jdbc, . URL https://www.javatpoint.com/java-jdbc.

[8] Jdbi, . URL https://jdbi.org/.

[9] Android jetpack. URL https://developer.android.com/jetpack.

[10] Json web token. URL https://jwt.io/.

[11] Android keystore. URL https://developer.android.com/training/articles/keystore.

[12] Kotlin. URL https://developer.android.com/kotlin.

[13] Ktor. URL https://ktor.io/.

[14] Rainbow tables. URL https://en.wikipedia.org/wiki/Rainbow_table.

[15] React framework. URL https://reactjs.org/docs/getting-started.html.

[16] Android sharedpreferences. URL https://developer.android.com/reference/android/content/SharedPreferences.

[17] Spring mvc, . URL https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html.

[18] Spring security, . URL https://spring.io/projects/spring-security.