



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

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 (Application 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;
- **DAO (Data Access Object)**: A pattern that provides an abstract interface to some type of database or other persistence mechanism;

Contents

1	Introduction	1
2	Project's context and objectives	3
2.1	Context	3
2.2	Real world comparision	4
2.3	Objectives	4
3	Components and requirements	5
3.1	Requirements analysis	5
3.1.1	Database	5
3.1.2	HTTP Server	5
3.1.3	Mobile application	6
3.1.4	Front web application	6
3.2	Component's structure	6
3.3	Stakeholders	7
3.3.1	User	7
3.3.2	External API	7
4	Project development	9
4.1	Relational database	9
4.2	Food API's	9
4.3	Android client	10
4.4	Restaurant APIs	10
5	Results	13
5.1	Relational database	13
5.1.1	Used tecnologies	13
5.1.2	Conceptual model	13
5.2	HTTP server	15
5.2.1	Used tecnologies	15
5.2.2	Code structure	15
5.2.3	Request lifecycle	16
5.2.4	JDBI	16
5.2.5	Spring Security	17
5.3	Geolocation	19
5.4	Android application	19
5.4.1	Used tecnologies	19
5.4.2	Code structure	20
5.4.3	Local data storing	21
5.4.4	User authentication and authorization	21

5.4.5	Data synchronization	22
5.4.6	Android version compatibility	23
5.4.7	Functionalities	23
5.5	Web browser application	23
5.5.1	Used tecnologies	23
5.5.2	Code structure	23
5.5.3	Functionalities	23
6	Conclusion	25
6.1	Future development	25
7	Appendices	27

List of Figures

Nutr.io platform components	7
An example of a restaurant's menu that would bring implications to the OCR translation . .	10
Database conceptual model	14
Server's classes structure	16
JDBI using an imperative API style	17
JDBI using a declarative API style (example from HTTP server)	17
The JWT workflow	18
A Spring security workflow example with the POST /user/login	18
The repository pattern diagram	20
API nutritional accuracy sheet	32

Chapter 1

Introduction

In the context of the undergraduate programme of science and computer engineering at the university of "Instituto Superior de Engenharia de Lisboa" for the course "Project and Seminary" [19], it is considered the development of a Bachelor's final project with the intended learning outcomes of applying skills acquired throughout the course to solve a problem, either individually or as a team, describing and testing the developed work, and defending technical solutions.

With this purpose, the elaboration of this project aims to describe and contextualize mentioned objectives and activities developed by the students which will compose not only as a grading tool, but also as a moment of critical reflexion and learning.

This project and its' work was developed under the tutorage of engineer Fernando Miguel Gamboa de Carvalho during the 2019/2020 summer semester of "Instituto Superior de Engenharia de Lisboa".

In the elaboration of this project, the students chose a field which could facilitate and contribute in the improvement of day-to-day activities of people with diabetes, and as such, developed a multi-platform system capable of providing users an accurate carbohydrate quantity for meals served in restaurants.

This document is structured in 6 distinct parts, with an appendix which contains additional information about a topic, having references pointing to it when necessary.

We will conclude with a final remark about future work and a reflexion regarding our academic course and lessons learned.

Chapter 2

Project's context and objectives

2.1 Context

Type 1 diabetes is caused by an autoimmune reaction where the body's defence system attacks the cells that produce insulin. As a result, the body produces very little or no insulin. The exact causes of this are not yet known, but are linked to a combination of genetic and environmental conditions. [6]

Healthy nutrition — knowing when and, most importantly, what to eat — is an important part of diabetes management as different foods affect your blood glucose levels differently. Foods with a carbohydrate count require an insulin dosage to be administered which is calculated based on an uniquely crafted insulin profile for the person with type 1 diabetes.

Knowing the carbohydrates of what is being eaten solely relies on mapping a food's portion (such as weight or cups) to its' nutritional value using an official nutritional sheet [17] or a nutritional application, meaning that with access to a food scale this task becomes simple. However, it is unrealistic to expect a person with type 1 diabetes to always bring a food scale to every scenario - such as a restaurant - leading to inaccurate estimations on eaten meals and as such, incorrect insulin dosages.

Given that one of the group members has type 1 diabetes and that according to IDF, an estimated 1.1 million children and adolescents under the age of 20 live with type 1 diabetes [7] we decided to focus on this subject and as such faced a problem:

Despite the fact that countless nutritional applications are capable of providing nutritional information, it is not accurate and trustworthy regarding complex meals and even when it is, this information is always isolated in context, meaning that said meal always has a generic portion and is served in a generic place.

2.2 Real world comparison

In order to consolidate and define the problem in relation to the project we researched various mobile nutritional applications, amongst them "MyFitnessPal" as the main comparison. This application allows users to search a meal and its' information by name, recipe, commercial barcode or geolocation. The latter only providing meals for popular fast food chains around the user.

The nutritional book and mobile application "Carbs and Cals" is also worth mentioning as it also provides pictures of searched meals, allowing the user to better understand and visualize the portions that given meal might have.

2.3 Objectives

Given the previously context and exemplification, this platform pretends to offer an application that searches for nearby restaurants and their meals, providing nutritional information about them such as glucose and carbohydrates amounts. This provided information becomes useful when calculating the insulin dosage, given the user's actual blood glucose and the selected meal, which is one of the main functionalities of this application.

This platform also aims to be self-maintainable, meaning that the user's submits and feedbacks will allow the platform to grow and make the submitted data more accurate.

Here are the five main objectives this project aims to fulfill by the end of development:

- 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.

Chapter 3

Components and requirements

3.1 Requirements analysis

In order to build this multiplatform 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.

3.1.1 Database

Functional requirements

- A model that stores and organizes information from different 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;
- Separate data, labeling which ones should be votable, favorable or reportable;

Non-functional requirements

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

3.1.2 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

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

3.1.3 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 minimum functionalities for unauthenticated users;

Non-functional requirements

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

3.1.4 Front web application

Functional requirements

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

Non-functional requirements

- Responsive and performant UI;

3.2 Component's structure

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



Figure 3.1: 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, we chose to use the Here API, to provide restaurant information and geolocation.

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

3.3 Stakeholders

Analyzed components and requirements imply the existence 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:

3.3.1 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.

3.3.2 External API

An external API interacts directly with the HTTP server and is responsible 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 4

Project development

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

4.1 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 regarding how meals would be handled.

4.2 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 [Appendix C - API nutritional accuracy sheet]

We assumed 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, we 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.

4.3 Android client

The Android application's development progressed normally but it had to be put on hold several times, 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.

4.4 Restaurant APIs

During early stages of development, we 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 and is not reliable in cases where restaurants design their menus with just an image of a meal and its' price.

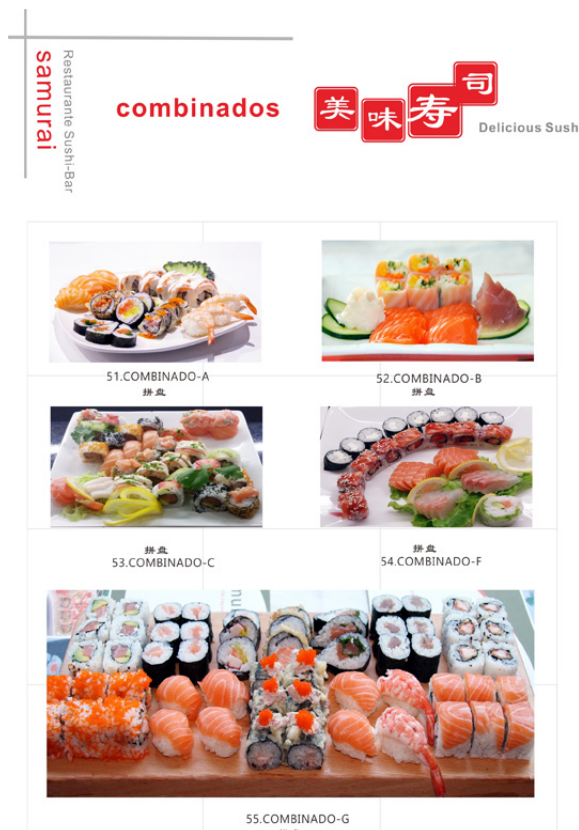


Figure 4.1: An example of a restaurant's menu that would bring implications to the OCR translation

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 5

Results

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

5.1 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.

5.1.1 Used technologies

When the project was in a planning phase, we 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[18] plugin, and is supported by Heroku - allowing to deploy the application in later stages of development.

5.1.2 Conceptual model

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

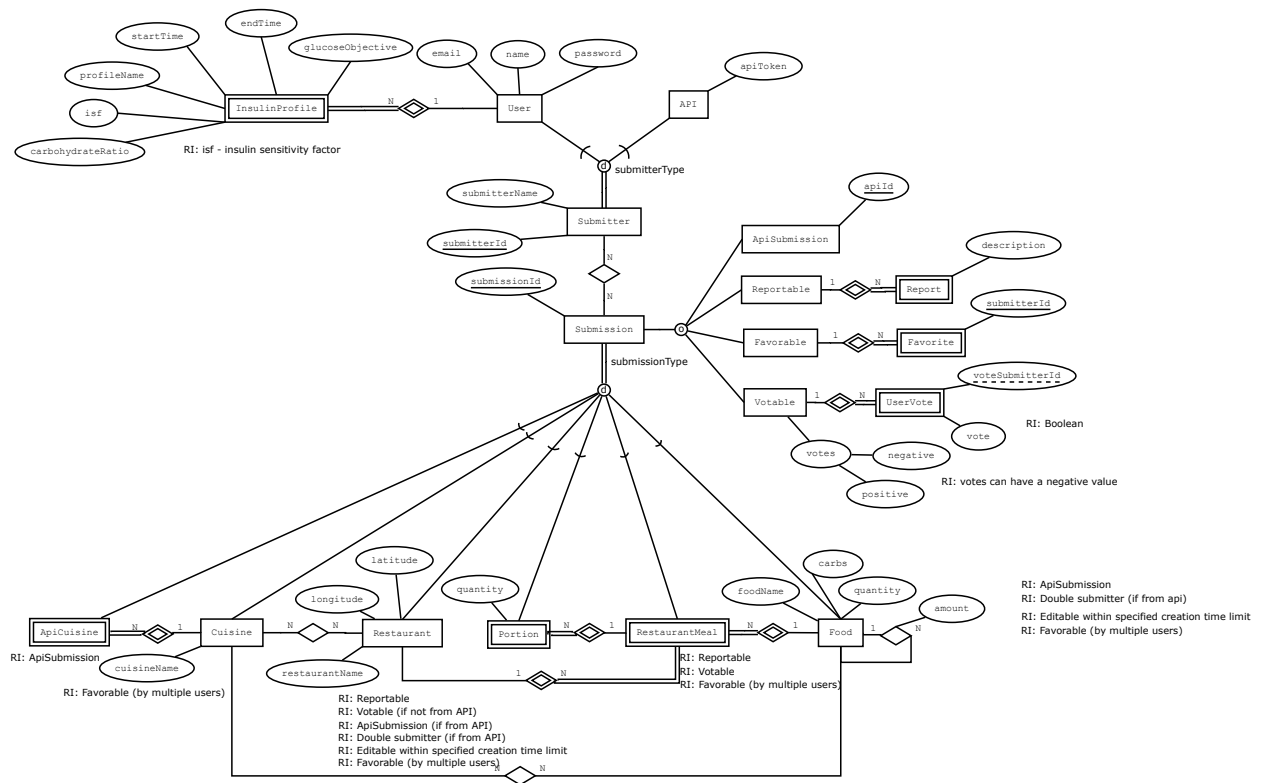


Figure 5.1: Database conceptual model

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

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

Now the submission can fall into 4 categories: ApiSubmission, Reportable, Favorable and Votable, in order to distinguish 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.

5.2 HTTP server

5.2.1 Used technologies

Kotlin

We chose to use Kotlin[15] 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[8].

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 we felt comfortable with.

Spring MVC

At the beginning of the project we decided to use Spring MVC[24] rather than Ktor[16], 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, we 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.

5.2.2 Code structure

The HTTP server uses a Spring MVC implementation, structuring its' code in several layers:

- MVC Controller layer, uses multiple services and handles input to model and model to output *DTO* mappings through *DTO* class mappers. Data Transfer Objects (*DTO*): For every model there is an equivalent input and output *DTO* class which is represented by its corresponding suffix, e.g. "*InsulinProfilesInput*" and "*InsulinProfilesOutput*". Controller classes are prefixed by "*Controller*".
- Service layer, uses multiple repositories and handles model to database *DTO* and database *DTO* to model mapping through instance mappers. The mapper classes used to map the *DTOs* to model have a domain prefix e.g. "*Db*" or "*Api*" while having a type "*Mapper*" suffix e.g. "*DbRestaurantMapper*". The service class names are prefixed by "*Service*".

- Data Access layer, is ranged between repositories and Data Access Objects (*DAOs*), each repository uses various *DAOs* that represent each database entity/relation through JDBC declarative API. For database *DTOs*, the prefix "*Db*" is used before the table name and then the suffix "*DTO*". For *DAOs* it was used the table name and the prefix "*Dao*" e.g. "*UserDao*". Repository classes are prefixed by "*Repository*" e.g. "*RestaurantDbRepository*".

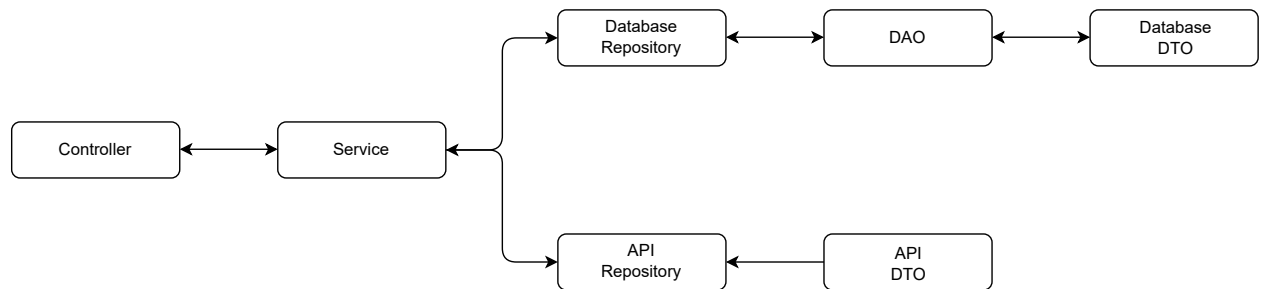


Figure 5.2: Server's classes structure

5.2.3 Request lifecycle

Each request is received in a spring controller mapping, with input object, parameters or path variables as arguments. When receiving an input object *DTO*, it will be validated with java's *javax.validation* (Hibernate). Once validated the input model's data might be used in subsequent service layer calls.

Repositories are used on the service level to handle all *DAO* interactions with the database and other HTTP API requests. The resulting responses (input and database *DTOs*) are converted to the model *DTOs* according to their response mappers. All API calls are made from asynchronous HTTP requests, while database calls block the calling thread.

5.2.4 JDBC

After some discussion of which driver should be used to allow communication between the server and database, we decided that the JDBC[11] was the best option as it is a library built on JDBC[10].

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

```
// 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 5.3: JDBC using an imperative API style

```
@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>
```

Figure 5.4: JDBC using a declarative API style (example from HTTP server)

We chose to use a declarative style due to its convenience and code simplicity over the imperative one. It was also the more adequate style to the project's structure.

5.2.5 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, we 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**[13] 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 a valid token is stolen, neither the attacker has a way to generate valid tokens nor does he know the user's password. The token could 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.

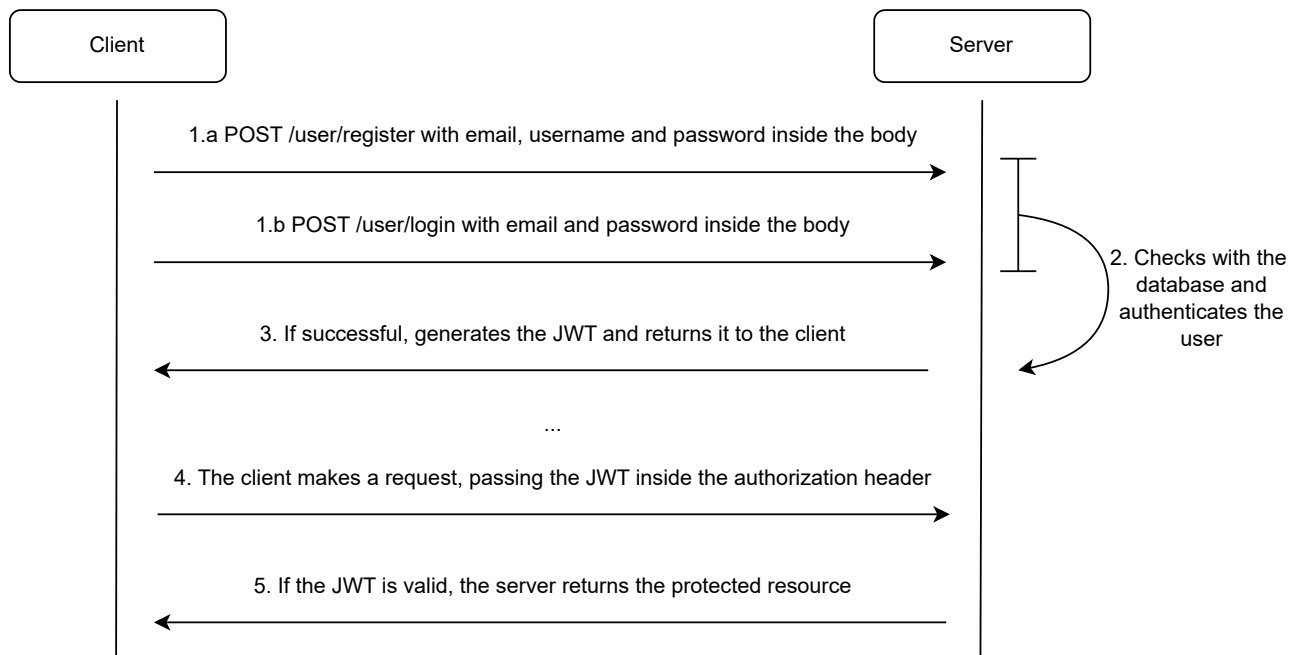


Figure 5.5: The JWT workflow

Implementation

To implement the shown workflow inside the HTTP server, as we are implementing it with Spring, the more obvious choice was to use **Spring Security**[25].

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

The picture below shows how the server handles an user login using this framework.

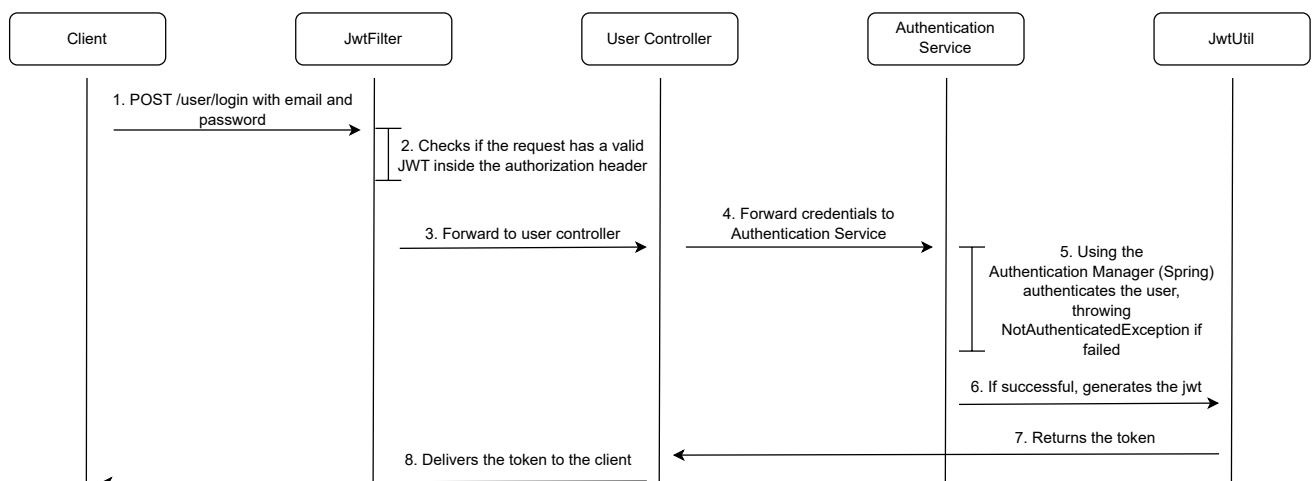


Figure 5.6: A Spring security workflow example with the POST /user/login

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 au-

thentication, 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. 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 guarantees 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**[1] before inserting the new user into the database.

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

- Already pre salts the passwords, preventing rainbow table attacks[20];
- 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.

5.3 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[5] 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, we 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.

5.4 Android application

5.4.1 Used technologies

Kotlin

We 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 optional 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.

5.4.2 Code structure

Drawing pattern

The mobile application code structure follows the **repository pattern**, which is a code architecture recommended by the **Android Jetpack**[12] for this type of applications.

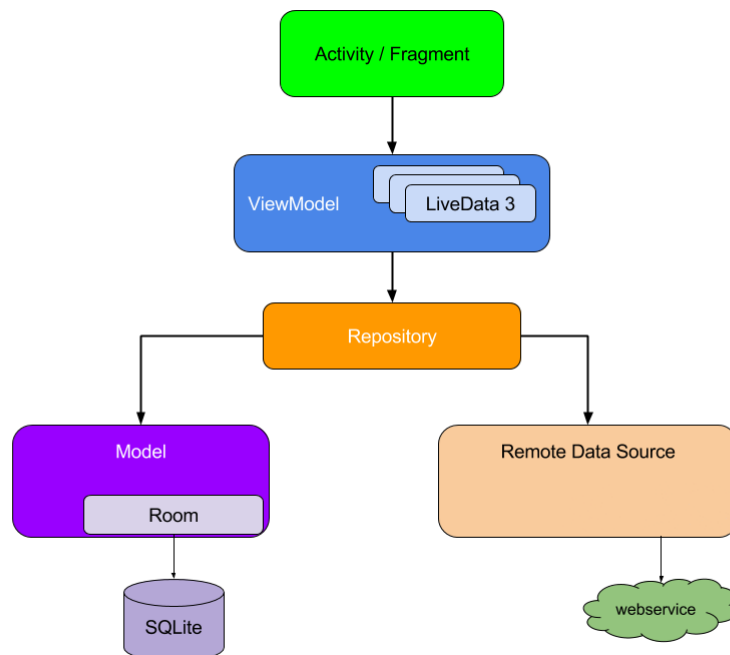


Figure 5.7: The repository pattern diagram

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.

Just as the server the android client has the input and output "DTOs" suffixed with "input" and "output". These classes are mapped by input mappers prefixed with "input".

The Activity, Fragments, ViewModel, Adapters and Repositories classes are all prefixed by their type.

Fragments

We 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 instantiate 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.

5.4.3 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.

Room classes use the "Db" prefixed followed by the name and prefixed by their type e.g. "DbMealInfoMapper", "DbMealInfoRelation" and "DbMealInfoEntity". The DAO classes However are only prefixed with "Dao". All entities are mapped by a database mapper suffixed with "Mapper".

5.4.4 User authentication and authorization

The user has the ability to register and login in the mobile application. Besides being the server responsible 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[23] 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[22] 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[14] 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.

We 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. We 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.

5.4.5 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.

5.4.6 Android version compatibility

In order to guarantee 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).

5.4.7 Functionalities

Here will be displayed pictures of the mobile application and its functionalities, explaining each one.

This section will be completed in the final release, once every UI aspect is resolved

5.5 Web browser application

5.5.1 Used technologies

React framework

We chose to build the website with JavaScript [9] using the React framework [21] , as it was the framework lectured in the Web applications development course and has innumerable advantages to other frameworks, such as Node.JS.

5.5.2 Code structure

Single-page application

As website design pattern, we 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

5.5.3 Functionalities

Under development

Chapter 6

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.

6.1 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 we 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, we 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 improvement 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 7

Appendices

This chapter displays all the appendices referenced in this report.

Appendix A - 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, apild
- Primary Key(s): submissionId, apild
- 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 B - 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 C - 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 kidney 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 1: 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_\(cipher\)](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] Haversine. URL https://en.wikipedia.org/wiki/Haversine_formula.
- [6] International diabetes federation, . URL <https://www.idf.org/aboutdiabetes/type-1-diabetes.html>.
- [7] Worldwide toll of diabetes, . URL <https://www.diabetesatlas.org/en/sections/worldwide-toll-of-diabetes.html>.
- [8] Java, . URL https://java.com/en/download/faq/whatis_java.xml.
- [9] Javascript, . URL <https://developer.mozilla.org/pt-PT/docs/Web/JavaScript>.
- [10] Jdbc, . URL <https://www.javatpoint.com/java-jdbc>.
- [11] Jdbi, . URL <https://jdbi.org/>.
- [12] Android jetpack. URL <https://developer.android.com/jetpack>.
- [13] Json web token. URL <https://jwt.io/>.
- [14] Android keystore. URL <https://developer.android.com/training/articles/keystore>.
- [15] Kotlin. URL <https://developer.android.com/kotlin>.
- [16] Ktor. URL <https://ktor.io/>.
- [17] Manual de contagem de hidratos de carbono na diabetes mellitus para profissionais de saúde. URL https://www.apn.org.pt/documentos/manuais/Manual_Contagem_HC.pdf.
- [18] Postgis. URL <https://postgis.net/workshops/postgis-intro/>.
- [19] Project and seminary. URL <https://isel.pt/en/subjects/project-and-seminary-leic>.
- [20] Rainbow tables. URL https://en.wikipedia.org/wiki/Rainbow_table.

- [21] React framework. URL <https://reactjs.org/docs/getting-started.html>.
- [22] Android rooting. URL [https://en.wikipedia.org/wiki/Rooting_\(Android\)](https://en.wikipedia.org/wiki/Rooting_(Android)).
- [23] Android sharedPreferences. URL <https://developer.android.com/reference/android/content/SharedPreferences>.
- [24] Spring mvc, . URL <https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html>.
- [25] Spring security, . URL <https://spring.io/projects/spring-security>.