



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;
- **Lazy loading**: A design pattern to defer initialization of an object until the point at which it is needed;
- **Eager loading**: The opposite of Lazy loading. The object's initialization always occurs even when it is not needed;
- **CLI** - Command-Line Interface: an environment that processes commands to a computer program in the form of lines of text;

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 technologies	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	Routing and endpoints	16
5.2.4	JDBI	17
5.2.5	Fetching collections of data and sequences	23
5.2.6	Request lifecycle	27
5.2.7	Restaurants	27
5.2.8	Pagination	28
5.2.9	Spring Security	29
5.2.10	User data safety	31
5.3	Geolocation	31

5.4	Android application	32
5.4.1	Used tecnologies	32
5.4.2	Code structure	32
5.4.3	Code hierarchy	34
5.4.4	Local data storing	36
5.4.5	User authentication and authorization	36
5.4.6	Android version compatibility	37
5.4.7	Functionalities	37
5.5	Web browser application	47
5.5.1	Used tecnologies	47
5.5.2	Code structure	47
5.5.3	Functionalities	49
5.6	Deployment	53
6	Conclusion	55
6.1	Future development	55
7	Appendices	57

List of Figures

Nutr.io platform components	7
An example of a restaurant's menu that would bring difficulties to the OCR translation . . .	10
Database conceptual model	14
Server's classes structure	16
The server's routing	16
Example of a model class in our server	18
Database connections required to build a restaurant model	19
Using RequestContextHolder to create and retrieve values	20
DatabaseContext using our static method to create and later close a handle	20
Explicitly resetting request context and its' resources	21
DatabaseContext inTransaction implementation	21
Example usage of a database connection	21
JDBI imperative style	22
JDBI declarative style - defining Data Access Objects	23
JDBI declarative style - Calling defined Data Access Objects	23
Example of a database operation usin jdbc declarative type in our application	23
Using ClosableSequence with JDBI's ResultIterable	24
Using MemoizedSequence example	24
Example of a sequence being terminally operated multiple times	25
Fetching restaurants from the API asynchronously	26
Searching nearby restaurants with parallel work	27
The JWT workflow	29
A Spring security workflow example with the POST /user/login	30
The repository pattern diagram	33
BaseViewModelFragment save and argument restore methods	35
IViewModelManager onSaveViewModels method	35
IParentViewModel's navParentViewModel function	36
Login and register workflow	38
The logout and account deletion fragment	39
User account deletion workflow	39
Restaurant detail	42
Meal selection menu	46
The web client's navigation diagram	48
Register and login page	49
Create a hardcoded meal or ingredient	49
Meal description (name, quantity and cuisines)	50
Ingredients (in the form of basic ingredients and other complex meals) are chosen	50
User chooses quantity for all chosen ingredients	51
Input confirmation	51

Send	52
Insulin profile creation	52
List of reports (restaurants and meals)	53
A reports detail	53
Environment variable change inside application.properties	54
Heroku deployment procedure	54
API nutritional accuracy sheet	61

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" [14], 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. [5]

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 [11] 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 [6] 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 comparison, 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[23], 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 difficulties 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 maintaining 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[24] plugin, and is supported by Heroku - allowing to deploy the application in later stages of development.

5.1.2 Conceptual model

Below 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.

A submission can fit 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 an association called ApiCuisine in order to map cuisine identifiers unique to the Here API to our own cuisines.

A food is always one of the following: a suggested meal, a custom meal or an ingredient - all identified by their submission type (SM, CM and I, respectively).

A meal can also be composed by other meals or ingredients, represented by the 1-N Food association.

Regarding security and encryption, all sensible user information such as medical records in the InsulinProfile table or passwords in the User table are encrypted or hashed.

More details about security can be found in [Chapter 8 - Encryption and Sensitive data]

5.2 HTTP server

5.2.1 Used technologies

Kotlin

We chose to use Kotlin[28] 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[31].

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[21] rather than Ktor[10], 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 Routing and endpoints



Figure 5.3: The server's routing

The picture presented above represents a simplified diagram that shows how the navigation between endpoints occurs. It should be taken in consideration that, as the picture's label says, some nodes represented in this diagram are used by both users and moderators and the diagram can not label those situations with detail. Those details can be better analysed in this report appendix about endpoints.⁷

The color code used in this diagram represents where each endpoint starts providing a better comprehension and visualization, e.g.: `/restaurant/:restaurantId/meal/:mealId` (endpoint for a specific restaurant meal) starts with a blue node (`/restaurant`); `/meal/suggested` (endpoint for suggested meals) starts with a yellow node (`/meal`).

5.2.4 JDBI

We decided to use JDBI as the library responsible for allowing communication between the server and the database due to teacher's recommendations, integration with Spring and Kotlin, and finally its' declarative functionalities, which are explained further on.

As a brief description, JDBI is, according to the documentation:

"Jdbi is built on top of JDBC. If your database has a JDBC driver, you can use Jdbi with it. Jdbi improves JDBC's rough interface, providing a more natural Java database interface that is easy to bind to your domain data types. Unlike an ORM, we do not aim to provide a complete object relational mapping framework - instead of that hidden complexity, we provide building blocks that allow you to construct the mapping between relations and objects as appropriate for your application."

Although the documentation provided by JDBI describes in great detail its' functionalities and how to utilize them, some key aspects and differences which are specific to the project are explained in the following sections.

Getting started

The Jdbi class is the main entry point into the library and each instance requires a DataSource connection, **where a single instance** is created in **our** DatabaseConfig class according to the application properties provided in the resources folder.

Afterwards, a database connection can open using the JDBI instance and is represented by JDBI's `Handle` class. According to the documentation:

"A handle is used to prepare and run SQL statements against the database, and manage database transactions. It provides access to fluent statement APIs that can bind arguments, execute the statement, and then map any results into Java objects."

Avoiding multiple connections and DatabaseContext

Considering the following example of a model from our server:

```

class Restaurant(
    identifier: Lazy<RestaurantIdentifier>,
    name: String,
    image: URI?,
    override val isFavorite: UserPredicate,
    override val isFavorable: UserPredicate,
    override val userVote: UserVote,
    override val votes: Lazy<Votes>,
    override val isVotable: UserPredicate,
    override val isReportable: UserPredicate,
    override val cuisines: Sequence<Cuisine>,
    val ownerId: Int?,
    val latitude: Float,
    val longitude: Float,
    val submitterInfo: Lazy<Submitter>,
    val creationDate: Lazy<OffsetDateTime?>,
    val meals: Sequence<Meal>,
    val suggestedMeals: Sequence<Meal>
) : BasePublicSubmission<Lazy<RestaurantIdentifier>>()

```

Figure 5.4: Example of a model class in our server

If we consider that the fields `votes`, `cuisines`, `submitterInfo`, `creationDate` and `meals` are always needed when fulfilling a HTTP request and obtained from the database, then we can also assume that throughout a single HTTP request, multiple database accesses can be performed, as seen in the following code for `DbRestaurantModelMapper`:

```

return Restaurant(
    identifier = lazy {
        val apiId : String? = submissionDbRepository.getApiSubmissionById(dto.submission_id)?.apiId
        RestaurantIdentifier(
            submissionId = dto.submission_id,
            apiId = apiId,
            submitterId = submitterInfo.value.identifier
        ) ^lazy
    },
    name = dto.name,
    ownerId = dto.ownerId,
    latitude = dto.latitude,
    longitude = dto.longitude,
    cuisines = cuisines,
    meals = dbMealRepository
        .getAllUserMealsForRestaurant(dto.submission_id)
        .map(dbMealMapper::mapTo),
    suggestedMeals = dbMealRepository
        .getAllSuggestedMealsFromCuisineNames(cuisines.map { it.name })
        .map(dbMealMapper::mapTo),
    votes = lazy { dbVotesMapper.mapTo(dbVoteRepository.getVotes(dto.submission_id)) },
    userVote = toUserVote { userId : Int -> dbVoteRepository.getUserVote(dto.submission_id, userId) },
    isVotable = { userId : Int? ->
        //A user cannot favorite on it's own submission
        submitterInfo.value.identifier != userId
    },
    isFavorite = toUserPredicate({ false }) { userId : Int ->
        dbFavoriteRepo.getFavorite(dto.submission_id, userId)
    },
    isFavorable = { userId : Int? -> //A user cannot favorite on it's own submission
        submitterInfo.value.identifier != userId
    },

```

Figure 5.5: Database connections required to build a restaurant model

This is done to support lazily acquired fields when building a model class in order to avoid querying large quantities of data and then not needing it.

Additionally, if we consider that said information for each field is obtained in its' own and new transactional context then this design can be performance heavy if we open multiple database connections as needed.

In order to avoid this, each HTTP request holds a single database connection if needed and closes it when completed, as we can assume that no more database data is needed afterwards.

This implementation is done by having every repository depend on a single instance of the DatabaseContext - our developed wrapper class which opens a new connection handle for each request that needs it, storing it in current request's context, made possible by Spring's RequestContextHolder[20], allowing for later closure when the request is fulfilled.

According to the documentation:

"Holder class to expose the web request in the form of a thread-bound RequestAttributes[19] object. The request will be inherited by any child threads spawned by the current thread if the inheritable flag is set to true."

With that in mind, the following static method was created in order to wrap creation and retrieval of values from RequestContextHolder:

```
fun <T : Any> getOrPut(key: String, valueSupplier: () -> T, callback: ((T) -> Unit)? = null): T {
    val requestAttributes : RequestAttributes = RequestContextHolder
        .getRequestAttributes()
        ?: CustomAttributes().also { RequestContextHolder.setRequestAttributes(it, inheritable: true) }

    @Suppress( ...names: "UNCHECKED_CAST" )
    var value : T? = requestAttributes.getAttribute(key, DEFAULT_SCOPE) as? T

    if (value == null) {
        value = valueSupplier()
        requestAttributes.setAttribute(key, value, DEFAULT_SCOPE)
    }

    if (callback != null) requestAttributes.registerDestructionCallback(key, { callback(value) }, DEFAULT_SCOPE)
    return value
}
```

Figure 5.6: Using RequestContextHolder to create and retrieve values

Which is then used by DatabaseContext when managing a Handle and closing it on request completion, using Spring's RequestAttributes.registerDestructionCallback(String name, Runnable callback, int scope).

```
@Component
class DatabaseContext(private val jdbci: Jdbi) {
    private val handle: Handle

    get() {
        return getOrPut(
            key = HANDLE_KEY,
            valueSupplier = { jdbci.open() },
            callback = { it.close() }
        )
    }
}
```

Figure 5.7: DatabaseContext using our static method to create and later close a handle

For specific cases where a database connection is done outside the context of a HTTP request (like the ApiSubmitterMapper class) and as such no request attribute is present, we developed an alternative class that implements RequestAttributes, named CustomAttributes. This implementation simply holds a map of values and requires whoever used it to explicitly remove it from current context and close underlying resources; as the example shows:


```

fun close() {
    handle.close()
    RequestContextHolder.resetRequestAttributes()
}

```

Figure 5.8: Explicitly resetting request context and its' resources

Given implementation avoids mentioned issues, however it has the consequence of having transactions depend on the HTTP request that triggered the database fetch, meaning that lengthy controller operations may lock database tuples for longer periods of time, depending on the used transactional isolation level.

Finally, given that JDBI's connection handle can't be closed after a transaction, every database access is also done with `DatabaseContext.inTransaction(Function<Handle, T>)` instead of JDBI's given methods that would close attached handle. Implementation and a typical use case is as follows:

```

fun <T> inTransaction(func: (Handle) -> T): T {
    if (!handle.isInTransaction) {
        handle.begin()

        try {
            val result :T = func(handle)

            handle.commit()
            return result
        } catch (e: Exception) {
            handle.rollback()
            throw e
        }
    }

    return func(handle)
}

```

Figure 5.9: DatabaseContext inTransaction implementation

```

fun getRestaurantMeal(restaurantId: Int, mealId: Int): DbRestaurantMealDto? {
    return databaseContext.inTransaction { it: Handle
        return@inTransaction it
            .attach(restaurantMealDao)
            .getByRestaurantAndMealId(restaurantId, mealId)
    }
}

```

Figure 5.10: Example usage of a database connection

Performing operations - Imperative and Declarative API

When a repository uses previously mentioned method `databaseContext.inTransaction(Function<Handle, T>)`, given argument function is responsible for performing a database operation with the open handle, either using an **imperative** implementation or a **declarative** one.

The JDBI documentation describes both the imperative and declarative style as follows, respectively:

"The Core API provides a fluent, imperative interface, which uses Builder style objects to wire up your SQL to rich Java data types."

"The SQL Object extension sits atop Core, and provides a declarative interface. Tell Jdbi what SQL to execute and the shape of the results you like by declaring an annotated Java *interface*, and it will provide the implementation."

Additionally, the JDBI documentation offers a simple query example using both styles:

```
List<User> users = jdbi.withHandle(handle -> {
    handle.execute("CREATE TABLE user (id INTEGER PRIMARY KEY, name VARCHAR)");

    // Inline positional parameters
    handle.execute("INSERT INTO user(id, name) VALUES (?, ?)", 0, "Alice");

    // Positional parameters
    handle.createUpdate("INSERT INTO user(id, name) VALUES (?, ?)")
        .bind(0, 1) // 0-based parameter indexes
        .bind(1, "Bob")
        .execute();

    // Named parameters
    handle.createUpdate("INSERT INTO user(id, name) VALUES (:id, :name)")
        .bind("id", 2)
        .bind("name", "Clarice")
        .execute();

    // Named parameters from bean properties
    handle.createUpdate("INSERT INTO user(id, name) VALUES (:id, :name)")
        .bindBean(new User(3, "David"))
        .execute();

    // Easy mapping to any type
    return handle.createQuery("SELECT * FROM user ORDER BY name")
        .mapToBean(User.class)
        .list();
});
```

Figure 5.11: JDBI imperative style

```
// Define your own declarative interface
public interface UserDao {
    @SqlUpdate("CREATE TABLE user (id INTEGER PRIMARY KEY, name VARCHAR)")
    void createTable();

    @SqlUpdate("INSERT INTO user(id, name) VALUES (?, ?)")
    void insertPositional(int id, String name);

    @SqlUpdate("INSERT INTO user(id, name) VALUES (:id, :name)")
    void insertNamed(@Bind("id") int id, @Bind("name") String name);

    @SqlUpdate("INSERT INTO user(id, name) VALUES (:id, :name)")
    void insertBean(@BindBean User user);

    @SqlQuery("SELECT * FROM user ORDER BY name")
    @RegisterBeanMapper(User.class)
    List<User> listUsers();
}
```

Figure 5.12: JDBC declarative style - defining Data Access Objects

```
// Jdbi implements your interface based on annotations
List<User> userNames = jdbi.withExtension(UserDao.class, dao -> {
    dao.createTable();

    dao.insertPositional(0, "Alice");
    dao.insertPositional(1, "Bob");
    dao.insertNamed(2, "Clarice");
    dao.insertBean(new User(3, "David"));

    return dao.listUsers();
});
```

Figure 5.13: JDBC declarative style - Calling defined Data Access Objects

In the context of our project, we use the declarative API due to its convenience and code simplicity over the imperative one. The following is an example of a typical database operation in our application:

```
fun getAllByMealId(mealId: Int): Sequence<DbCuisineDto> {
    return databaseContext.inTransaction { handle :Handle ->
        handle.attach(CuisineDao::class.java)
            .getByMealId(mealId)
            .asCachedSequence()
    }
}
```

Figure 5.14: Example of a database operation using jdbi declarative type in our application

5.2.5 Fetching collections of data and sequences

When it comes to fetching collections of data (such as restaurants, meals, etc.) from either the database or an API, said data can be obtained and iterated in an **eager** approach or a **lazy** approach.

We decided to implement the fetching of data in a lazy manner using Kotlin's Sequences[17] instead of Java 8 Streams[18] due to their additional iterating operations and easier caching mechanisms over Streams due to the "once of" nature of Streams (meaning that any call to a

terminal operation closes the stream, rendering it unusable).

Implementing Closable and Cached Sequences

For cases where a collection of data also holds additional resources attached that require to be closed, such as JDBC's `ResultIterable` and `ResultIterator` classes (explained in the following section), a `Sequence` that closes attached resources when terminally operated or when iterated with *Kotlin's* `use` (the equivalent to Java's `try-with-resources`) was needed.

This logic was implemented with the `ClosableSequence` class and extension method `asClosableSequence()`, which can be invoked on any object that implements `Iterator` and by providing a `onClose` argument or have given `Iterator` also implement `Closable`. A typical usage of this class can be described with the following example:

```
fun getAllUserFavorites(submitterId: Int, count: Int?, skip: Int?): ClosableSequence<DbRestaurantMealDto> {  
    return databaseContext.inTransaction { handle : Handle ->  
        return@inTransaction handle.attach(restaurantMealDao)  
            .getAllUserFavorites(submitterId, count, skip)  
            .asClosableSequence()  
    }  
}
```

Figure 5.15: Using `ClosableSequence` with JDBC's `ResultIterable`

However, this implementation does not support caching (also known as memoization), rendering it impossible to iterate its' values again after a terminal operation is done. To solve this, the extension method `memoized()` was created (in the `MemoizedSequences.kt` class) and can be invoked in any object that implements a `Sequence`. Seeing as this operation has additional overhead, it is only invoked when absolutely necessary, such as the following example, which terminally operates all restaurants obtained by the API multiple times:

```
val databaseRestaurants : Sequence<Restaurant> = dbRestaurantRepository  
    .getAllByCoordinates(latitude, longitude, chosenRadius, skip, count: count / 2)  
    .map(restaurantModelMapper::mapTo)  
    .memoized()  
  
return filterRedundantApiRestaurants(databaseRestaurants, apiRestaurants)
```

Figure 5.16: Using `MemoizedSequence` example

```

private fun filterRedundantApiRestaurants(
    dbRestaurants: Sequence<Restaurant>,
    apiRestaurants: Sequence<Restaurant>
): Sequence<Restaurant> {

    //Filter api restaurants that already exist in db
    val filteredApiRestaurants :Sequence<Restaurant> = apiRestaurants.filter { apiRestaurant : Restaurant ->
        //Db does not contain a restaurant with the api identifier
        dbRestaurants.none { dbRestaurant : Restaurant ->
            //Same apiId
            apiRestaurant.identifier.value.apiId == dbRestaurant.identifier.value.apiId
            //Same API
            && apiRestaurant.identifier.value.submitterId == dbRestaurant.identifier.value.submitterId
        }
    }

    return dbRestaurants.plus(filteredApiRestaurants)
}

```

Figure 5.17: Example of a sequence being terminally operated multiple times

Finally, when mentioned above that "Sequences allow for easier caching mechanisms when compared to Streams due to their 'once of' nature", if we consider the caching implementation designed in the course MPD (and also described here[27]), using memoized Streams would instead return `Supplier<Stream<T>` and therefor introduce additional overhead and variables.

Database fetching

According to the JDBC documentation[7], a query method supports the return of lazy collections using **Streams**, **ResultIterables** and **ResultIterators**:

"Query methods may also return a `ResultIterable`[15], `ResultIterator`[16], or a `Stream`[22]."

"As long as your database supports streaming results (for example, PostgreSQL will do it as long as you are in a transaction and set a fetch size), the stream will lazily fetch rows from the database as necessary."

Since no native support for Sequences exist, every DAO class returns a `ResultIterable` of data and calling repository is then responsible of mapping it to a Sequence.

However, this approach is not enough on its' own as JDBC warns that:

"`ResultIterable`, `ResultIterator` and `Stream` methods do not play nice with on-demand SQL Objects. Unless the methods are called in a nested way, the returned object will already be closed.

Given the following, it is simply not enough to have every *DAO* and *Repository* class return a lazy collection after a handle/transaction call, as the returned collection would be closed; so a solution was needed.

An initial solution involved opening a connection handle to fetch a Sequence of data and closing said handle when a terminal operation is executed, meaning that all data was safely extracted and cached. This solution proved to not be programmer friendly and with dangerous pitfalls, and as such was discarded since nothing "forced" the programmer to perform a terminal operation

which would lead to resource leakage.

Another alternative is to wrap a repository call in a nested way, as the JDBI documentation suggests. This approach relies on leaking database logic and dependencies to Controller and/or Service classes, which goes against a clean dependency injection design and as such was also discarded.

Finally, the adopted solution ties perfectly with the issues and developed solutions mentioned previously in chapter **5.2.4 JDBI** regarding "Avoiding multiple connections and DatabaseContext". If we assume that a Sequence and its' resources are no longer needed after the underlying HTTP request is closed, then having the DatabaseCleanupInterceptor class close the database connection avoids resource leakage when given sequences are not operated in a terminal way and having Controller and Service classes rely on database related logic.

API fetching

Considering that Spring MVC does not support replying content to the requester asynchronously, all data fetched from the APIs is done in a blocking way, with the exception of requesting nearby restaurants from the API, which is done asynchronously, as the image shows:

```
override fun searchNearbyRestaurants(
    latitude: Float,
    longitude: Float,
    radiusMeters: Int,
    name: String?,
    skip: Int?,
    count: Int
): Sequence<RestaurantDto> {
    val apiCount :Int = if (skip == null || skip == 0) count else count.plus(count.times(skip))
    if(apiCount > MAX_HERE_ITEMS) {
        return emptySequence()
    }
    return sequence { this: SequenceScope<RestaurantDto>
        val uri :URI = restaurantUri.nearbyRestaurants(latitude, longitude, radiusMeters, name, skip, apiCount)
        val response : HttpResponse<String>! = httpClient.send(
            buildGetRequest(uri),
            HttpResponse.BodyHandlers.ofString()
        )
        yieldAll(handleNearbyRestaurantsResponse(response).drop(0, apiCount - count))
    }
}
```

Figure 5.18: Fetching restaurants from the API asynchronously

This exception allows for time optimizations when searching for nearby restaurants as it requires results from both the database and the API meaning that parallel work can be executed, resulting in the following code:

```

fun getNearbyRestaurants(
    latitude: Float,
    longitude: Float,
    name: String?,
    radius: Int?,
    skip: Int?,
    count: Int
): Sequence<Restaurant> {
    val chosenRadius : Int = if (radius != null && radius <= MAX_RADIUS) radius else MAX_RADIUS

    //Get API restaurants
    val apiRestaurants : Sequence<Restaurant> = hereRestaurantApi
        .searchNearbyRestaurants(latitude, longitude, chosenRadius, name, skip, count / 2)
        .map(restaurantModelMapper::mapTo)

    val databaseRestaurants : Sequence<Restaurant> = dbRestaurantRepository
        .getAllByCoordinates(latitude, longitude, chosenRadius, skip, count / 2)
        .map(restaurantModelMapper::mapTo)

    return filterRedundantApiRestaurants(databaseRestaurants, apiRestaurants)
}

```

Figure 5.19: Searching nearby restaurants with parallel work

Taking advantage of Kotlin's Sequences and its `yieldAll` method, acquiring API results is only blocking when the sequence is iterated and the request's `CompletableFuture` is not yet completed. It is also worth mentioning that parallelism order of the two functions is not reversed (i.e: The database call is asynchronous while the API one isn't) for two reasons: Java's `HttpClient` library support for asynchronous calls and the fact that database connections **must** be done in the same thread as the `DatabaseCleanupInterceptor`, which is explained in the previous subsection: **5.2.4 JDBI - Avoiding multiple connections and DatabaseContext**.

5.2.6 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.

5.2.7 Restaurants

Unlike other submissions, restaurants can be obtained from two sources of truth - APIs and the database - creating obstacles that needed to be solved and explained in this section, such as data consistency and defining unique identifiers.

Data consistency

Restaurants provided by APIs are only copied to the database when new metadata is added (such as a user vote, meal or favorite) in order to avoid creation of unnecessary and duplicate tuples. After a restaurant is copied to the database with new metadata, the server ensures that

subsequent requests always return the database submission and not the API one by prioritizing database results.

When searching nearby restaurants by location, this is done by querying both the database and the API and removing API results if their identifier already exists in the database result. When requesting a specific restaurant, the server prioritizes searching the database first and only the API afterwards, if no initial result was found.

Unique identifiers

Given that restaurants can be obtained from two sources of truth, then we can consider that each source provides its' own unique identifier. A mechanism that merges identifiers needed to be implemented in order to avoid not knowing which source contains requested restaurant with identifier "ID1" or worse - having two different restaurants from different sources with the same identifier.

Said mechanism is *RestaurantIdentifier*, which always contains the submitter in order for the server to know from which source the restaurant is from, and may contain a database submission identifier and/or an API identifier. Additionally, when an API restaurant is copied to the database, its' *RestaurantIdentifier* will now contain both API and database identifier.

When a request is done with an outdated *RestaurantIdentifier* (one that represents a restaurant which is contained in the database but has no respective identifier), the server always queries the database first under the submitter and API submission pair in order to ensure data consistency. This behavior can be observed in the *RestaurantServiceTests* class.

5.2.8 Pagination

Some endpoints return lists to the clients, and as such these endpoints should not return all the items available that are associated to them in order to avoid the server from fetching big quantities of data, high transfer times of said data to the client and high memory usage on the client side.

To solve this issue we decided to implement a basic pagination mechanism, by implementing features in the following layers:

Controller

For endpoints that have pagination, two optional query parameters were added: skip and count. These two, for most cases, have a default value which is used when the client does not specify these parameters and, when they do, said values are also validated according to the following constraints:

- Skip and count must not be negative;
- Count must not be higher than the server maximum defined value: currently 40 items.

DAO

Database pagination is achieved using PostgreSQL LIMIT and OFFSET keywords.

The first one specifies how many items a result should hold and its' value is equal to count, while the second one represents how many objects should be skipped and as such, its' value is count parameter multiplied by the skip parameter.

5.2.9 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**[29] 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.

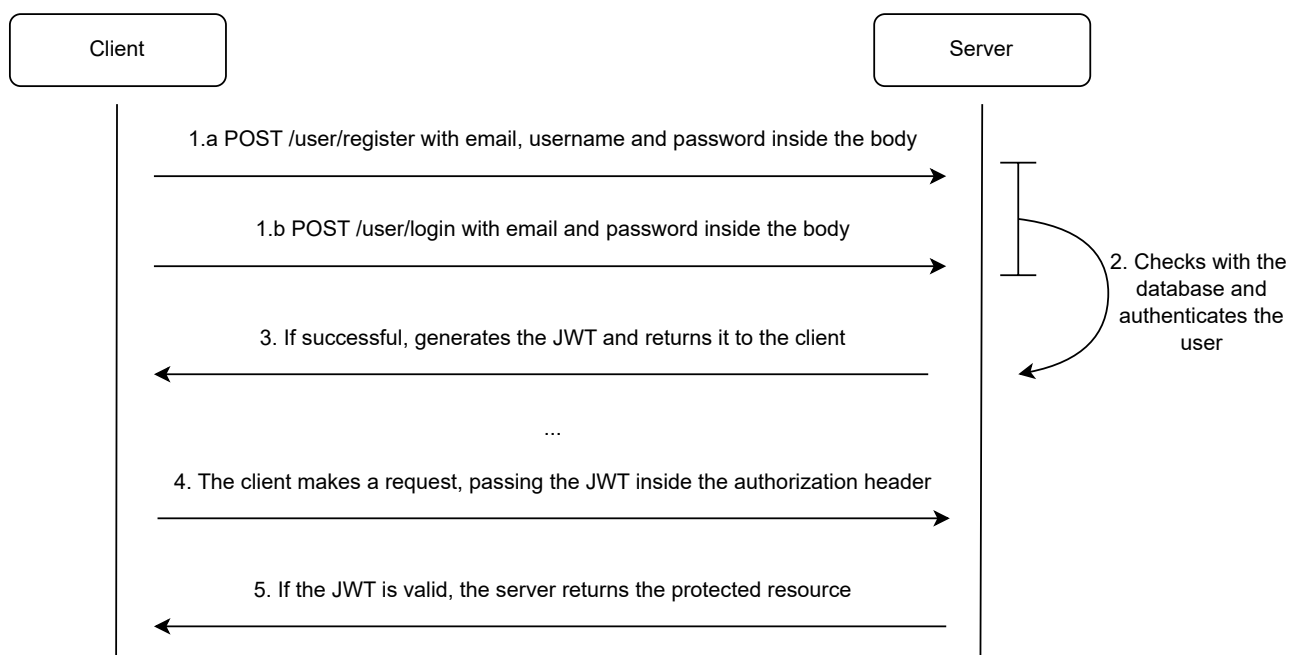


Figure 5.20: The JWT workflow

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**[30].

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

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

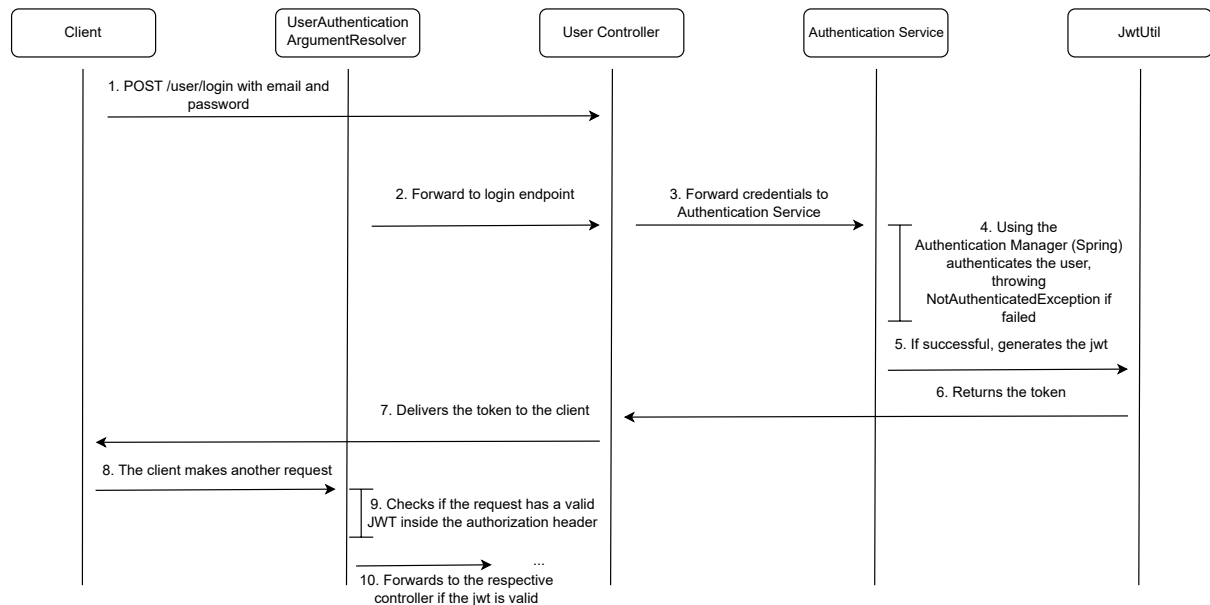


Figure 5.21: A Spring security workflow example with the POST /user/login

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. The JWT filter is also started up inside this class.

The `UserAuthenticationArgumentResolver` class, as the name says, is an argument resolver that filters each request, acting before it reaches the desired controller, checking the authentication header and extracting the jwt from the Bearer verifying if it is valid.

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**[33] before inserting the new user into the database.

BCrypt is a password-hashing function based on the Blowfish[34] cipher. We found this function very convenient for these reasons:

- Already pre salts the passwords, preventing rainbow table attacks[32];

- 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.

5.2.10 User data safety

In the previous section it was shown that the user can register an account. It was made this way so the server can store insulin profiles, which will be better explained in the mobile application's section. The insulin profiles store user sensitive information and are encrypted when written inside the database using a symmetric cipher (AES-256) with a private key provided by an environment variable from the host machine.

The user can also delete its account. When an account is deleted all user sensitive information is deleted from the server: email, username, password, insulin profile and custom meals. However, its submitter identifier, which is a database generated number, is maintained in order to preserve its public submissions, in order to keep the platform's data consistent, so a user that makes many public insertions does not compromise our data's consistency if its account is deleted.

This way we also comply to the General Data Protection Regulation (GDPR)[3], as we are keeping data safe by encrypting it into the database and we are not collecting user's data for other interests and, the most import of all, we provide the user the right to be erased from our platform removing every personal detail from it when an account deletion happens.

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[35] 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;
- **FlexBox** - a library which adds the flexible box layout, in order to hold multiple items inside the same box;
- **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**[8] for this type of applications.

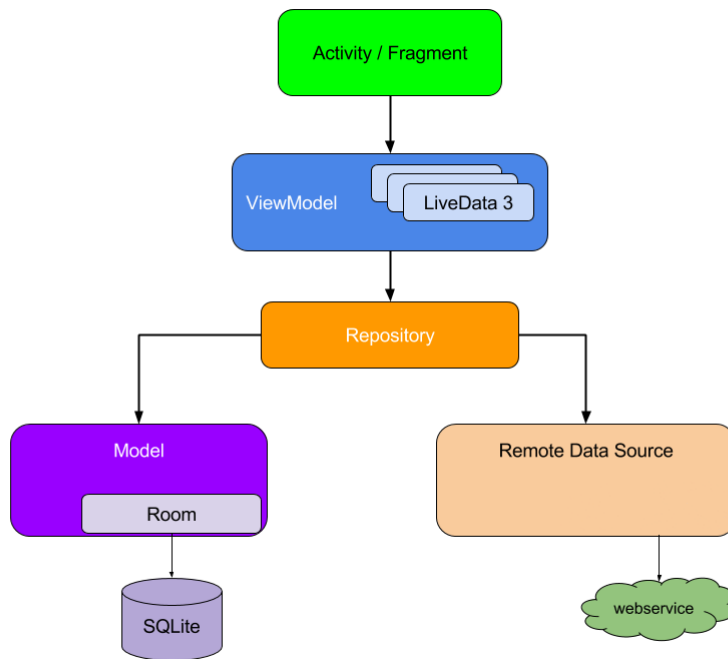


Figure 5.22: The repository pattern diagram

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[2] 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.

Modular interfaces

As the code in the mobile application development became repetitive, the group decided to implement modular interfaces, which are interfaces that can be implemented by fragments and viewholders and provide them predefined behaviours.

5.4.3 Code hierarchy

Fragments and their respective viewmodels, alongside with ViewModelProviders inherit from base classes that have already predefined code to avoid repetition.

ViewModels

Inside the ViewModel hierarchy there is the **BaseViewModel**, allowing item list restores through parcelable implementation, saving also pagination data.

BaseListViewModel also has **LiveDataListHandler** which handles all code related to live-data manipulation, such as: mapping and data restoration.

ViewModels that have `info` included in its name, know how get a detail from a specific resource (restaurant or meal) and extend from **BaseListViewModel**, storing subresources in its parent **LiveDataListHandler** and storing its own resource in its individual **LiveDataHandler**.

There are also **ItemPickerViewModels**, which extends from **BaseItemPickerViewModel** that has a separate **LiveDataListHandler** from its parent and it's used to store picked items, which are items that are picked inside the parent's list.

The items are moved between the parent and child lists using the ViewModel's `pick()` or `unpick()` method.

Fragments

The previous explained hierarchy also exists in fragments, with the following characteristics:

The **BaseFragment**, as the name says, is the fragment on which every other fragment is based on, being widely used for event logging and utility methods needed by child fragments.

The **BaseListViewModelFragment** manages the ViewModel and Fragment arguments' lifecycle, implementing `IViewModelManager`, from the modular interfaces, which supplies the code to build ViewModels. The fragment also has a ViewModel that is lazily created, using the previously saved instance state and can save and restore the ViewModel and Fragment arguments.

```

override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)

    //Save fragment arguments
    outState.putBundle(javaClass.simpleName, arguments)

    //Save fragment ViewModels
    super.onSaveViewModels(outState, getViewModels())
}

private fun restoreArguments() {
    val savedInstanceState : Bundle? = savedInstanceState
    if(savedInstanceState != null) {
        arguments = savedInstanceState.getBundle(javaClass.simpleName)
    }
}

open fun getViewModels(): Iterable<Parcelable> = listOf(viewModel)

```

Figure 5.23: BaseViewModelFragment save and argument restore methods

```

fun onSaveViewModels(outState: Bundle, viewModels: Iterable<Parcelable>) {
    log.v( msg: "Saving ViewModels (${viewModels.joinToString( separator: ",") { it: Parcelable
        it.javaClass.simpleName
    }})"
    viewModels.forEach { viewModel ->
        outState.putParcelable(viewModel.javaClass.simpleName, viewModel)
    }
}

```

Figure 5.24: IViewModelManager onSaveViewModels method

Not all fragments are limited to just one ViewModel and can also have create, use and restore other ViewModels through the same ViewModelProviderFactory.

The **BaseListFragment** extends from **BaseViewModelFragment** and its ViewModel extends from **BaseListViewModel**. This fragment has the **RecyclerHandler** which handles recycler list behaviours and objects, such as: loading and related UI aspects, the recycler adapter and the **ViewModel**.

It also inits the Recycler's **Scroll Listener** for pagination proposes. The pagination routine delegated to this listener is defined through a class called **AppScrollListener**.

The **BaseSlideScreenFragment** extends from **BaseFragment** and its the base fragment responsible for layouts with tabs, often used inside the application. This fragment has the objects required for a Tab layout: the TabAdapter, the TabLayout and the ViewPager (which supplies and manages the lifecycle of each page). Shortening the explanation: it is a fragment that holds multiple fragments, which are designated as tabs by this layout.

Fragments with navigation dependencies

Some fragments in our application require data from fragments created by them, as example - AddCustomMealFragment and its child SelectMealsSlideScreenFragment. This implies that a communication channel exists between them and this is achieved by using Navigation's `navGraphViewModel[12]` function to share a ViewModel between them, using a nested navigation id[13].

Some instances present in the code, reuse the same child fragment to obtain this fragments, for example: SelectMealsSlideScreenFragment which can have more than one parent fragment. To solve this problem we used the child fragment arguments to pass the nested navigation id, so it could build its ViewModel. As this mechanism was used multiple times (AddProfileFragment) we implemented the `navParentViewModel` function to automate this process - use `navgraph Id` stored in fragment arguments to get the shared ViewModel.

```
interface IParentViewModel : IViewModelManager

inline fun <reified VM : ViewModel, F> F.navParentViewModel(): Lazy<VM> where F : Fragment, F : IParentViewModel {
    return lazy {
        val parentNavigation : Navigation = requireNotNull(arguments?.getParentNavigation()) {
            "${javaClass.simpleName} Must have a parent navigation for navGraphViewModel!"
        }
        val viewModelLazy: VM by navGraphViewModels(parentNavigation.navId) {
            VMProviderFactorySupplier(arguments, savedInstanceState, requireActivity().intent)
        }
        viewModelLazy ^lazy
    }
}
```

Figure 5.25: IParentViewModel's navParentViewModel function

5.4.4 Local data storing

As mentioned in the dependencies, the mobile application utilizes Room to store data locally. This is convenient to allow using the application in offline situations.

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

5.4.5 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[18] 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[36] can easily access the shared preferences file and retrieve plaintext from it, which would compromise the user security.

Resolution: Androidx Crypto

The Androidx crypto[1] 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[9] 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.

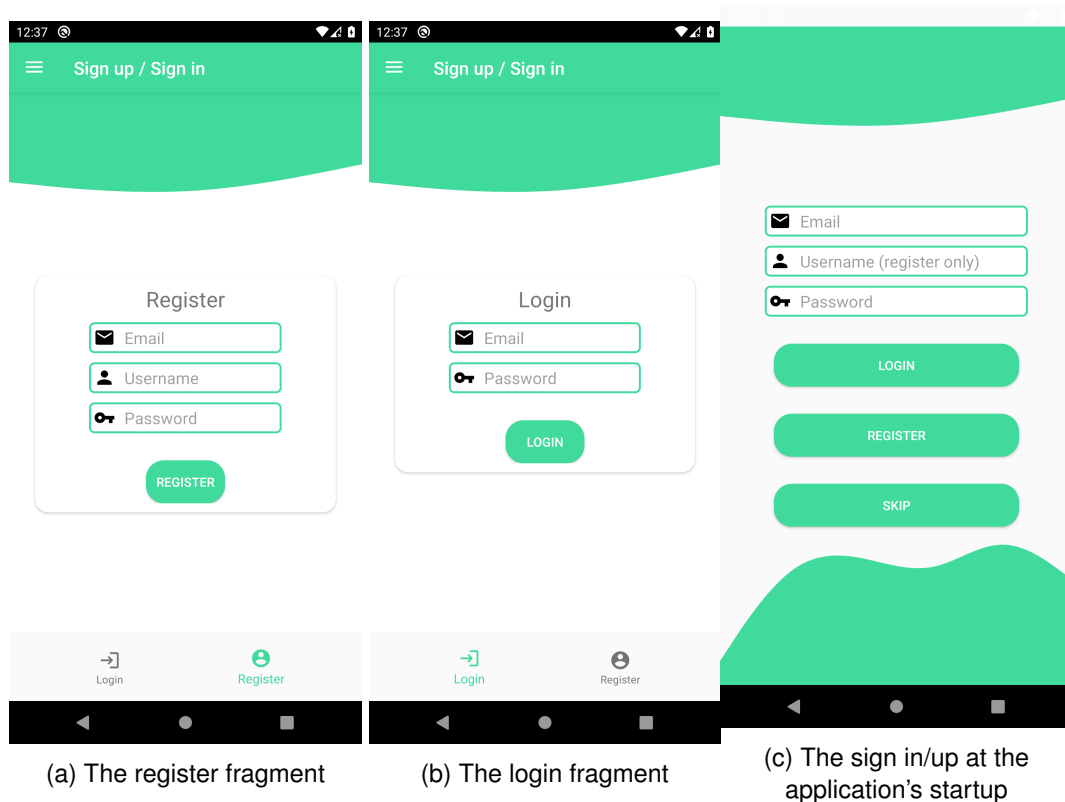
5.4.6 Android version compatibility

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.

Register and Login



Here are the 3 ways a user can register or login inside the mobile application: either on the application's startup or by clicking on the user's profile picture inside the drawer menu and accessing the slide screen with the bottom bar.

Here's the login and register workflow:

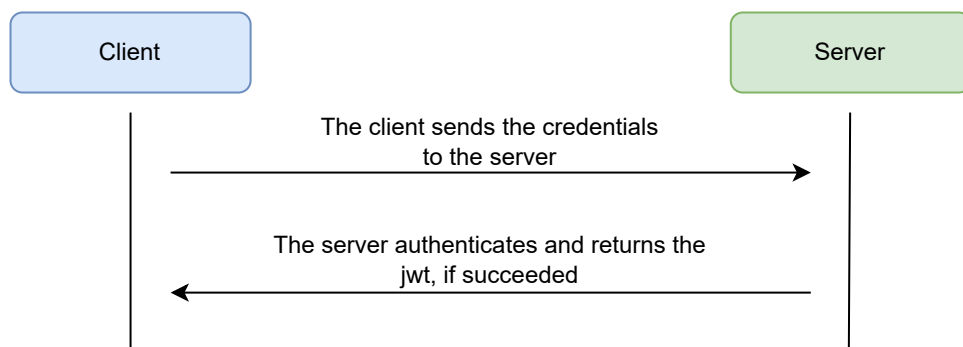


Figure 5.27: Login and register workflow

Account deletion

If the user goes to the login or register fragment again after being successfully logged in, it will see the fragment presented below.

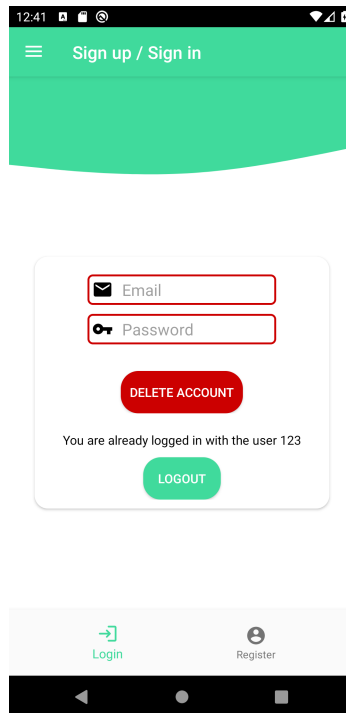


Figure 5.28: The logout and account deletion fragment

In order to delete the account, the user needs to fill the shown form again and then press the delete account button.

After that, the operation will occur as the diagram presented below.

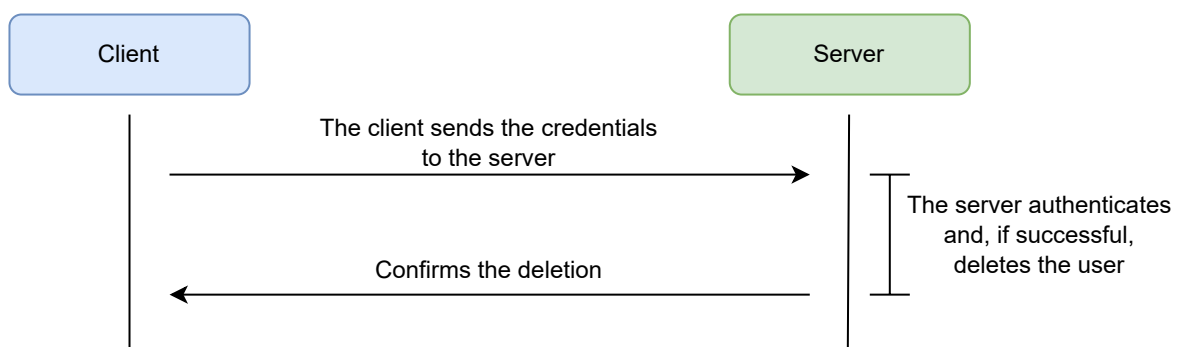


Figure 5.29: User account deletion workflow

After this only the submitter identifier will remain in database to preserve public submission and all sensitive data will be deleted.

Insulin profiles' creation and access

The image shows two screenshots of a mobile application interface. Screenshot (a) on the left is titled 'Add a profile to your routine'. It features a green header bar with a hamburger menu icon and the title. Below the header, there is a text input field containing 'Afternoon1'. A section titled 'Define the profile time period' contains two buttons, 'START TIME' and 'END TIME', with the values '16:46' and '19:46' respectively. Below this, there is a 'Glucose unit' dropdown menu set to 'mmol / L', a 'Your glucose objective' input field with the value '110', a question 'How much glucose is reduced by 1 insulin dose?' with an input field containing '40', and another question 'How many carbohydrates are lowered by 1 insulin dose?' with an input field containing '12' and a unit dropdown set to 'gr'. At the bottom is a 'CREATE' button. Screenshot (b) on the right is titled 'Insulin Profiles'. It shows a card for 'Afternoon1' with the following details: 'Starts at: 16:46', 'Ends at: 19:46', 'Glucose objective: 110.0 mmol / L', 'Insulin sensitivity factor: 40.0 mmol / L / insulin unit', and 'Carbohydrate ratio: 12.0 gr / insulin unit'. A green circular button with a white plus sign is located at the bottom right of the screen.

(a) The fragment to add an insulin profile

(b) The fragment to access and create insulin profiles

The user can map its day with insulin profiles, specifying for a certain timespan its glucose objective, insulin sensitivity and carbohydrates sensitivity, as this parameters change along the day.

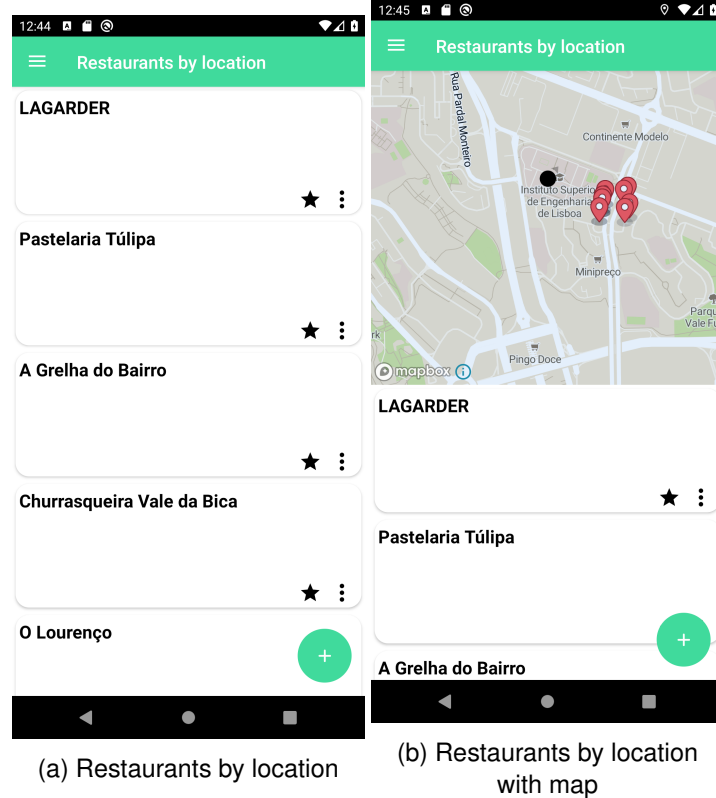
When creating insulin profiles, the user must know that a profile's time period can not overlap another and that the time mapping must be done from 00h00 to 23h59, meaning that the end time can not be before the start time.

If the user is not logged in, all insulin profiles created and deleted will be written inside the application's local database - Room. If the user is logged in, the profiles are written into our remote database.

Searching for restaurants and meals

The user can search for restaurant in two ways:

- By accessing 'Map view' inside the Restaurant Box, which will display a list of nearby restaurants along with a map;
- By accessing 'By name' inside the Restaurant Box, which will only display a list of restaurants which are also based on geolocation.



Restaurants can also be created by a user and added to a certain location, more specifically the current user's geolocation when the restaurant is being created. This can be done by clicking on the plus icon in the images above.

When clicking on a restaurant, the user can observe its menu. However if no submissions were made for this restaurant the menu will be filled with suggested meals from the database, as shown below.

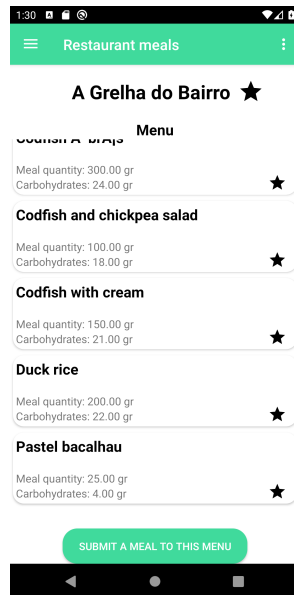
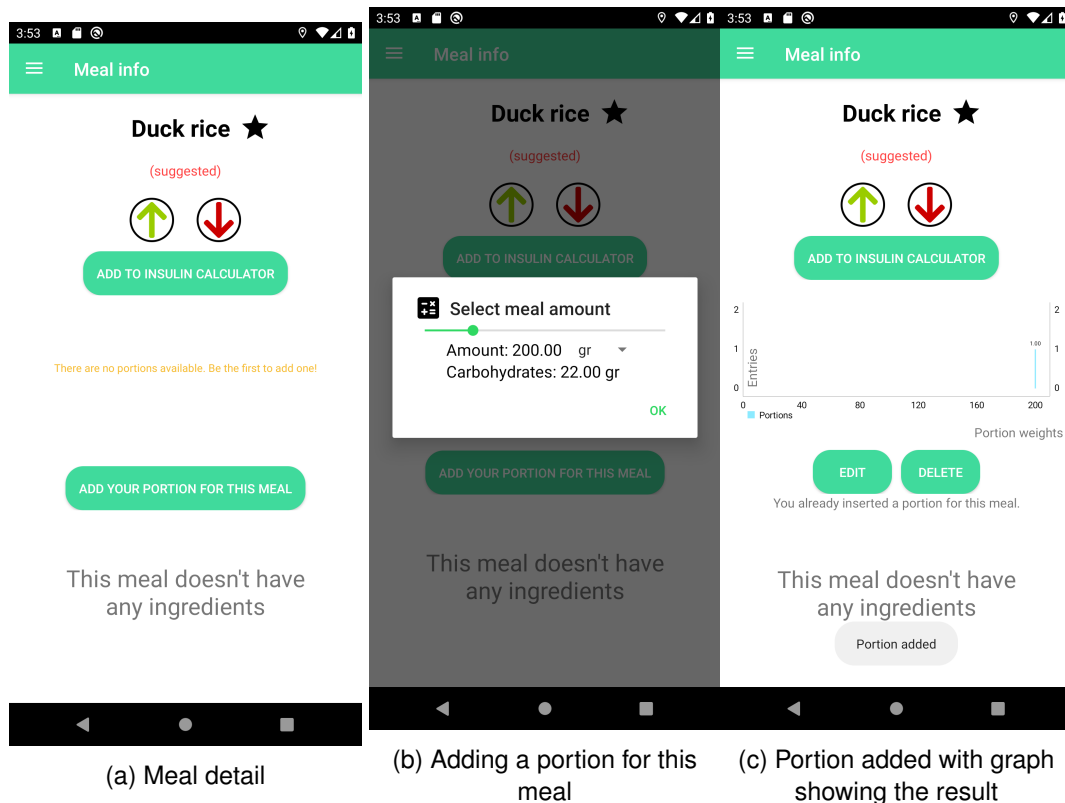


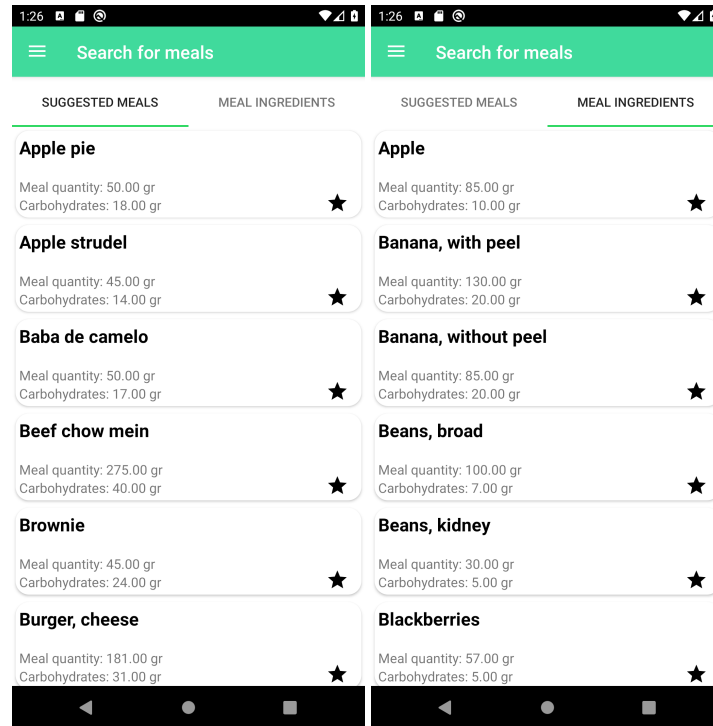
Figure 5.32: Restaurant detail

If a meal that exist in the restaurant's menu is not present in this list, the user can add it by clicking on the 'submit a meal for this menu' button.

If a certain meal in this list exist in the restaurant's menu, the user can add a portion by clicking on the meal inside the list and following this procedure:



The user can also access available meals by accessing 'By name' inside the Meals box, which will display a tab menu with the platform's suggested meals and meals' ingredients.



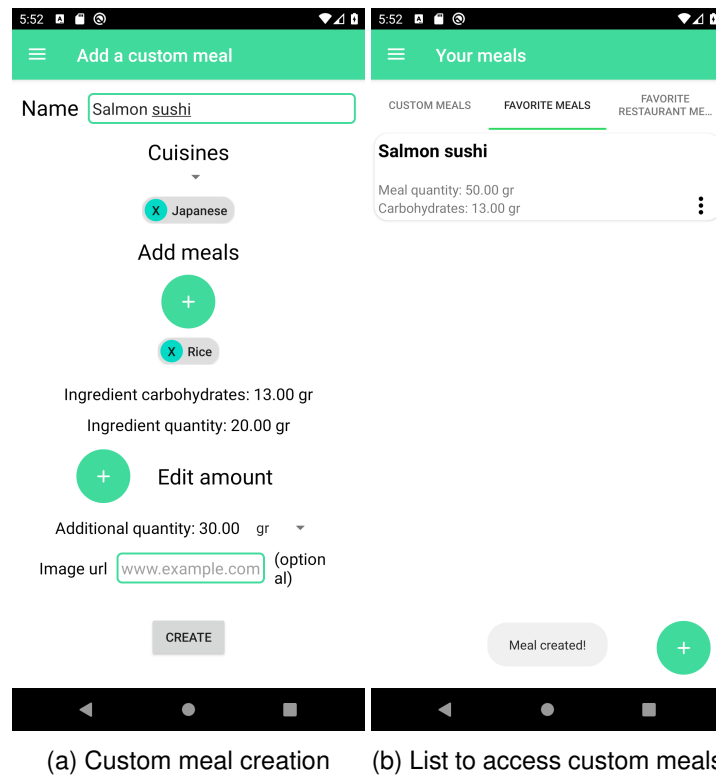
(a) Suggested meals list

(b) Meal ingredients list

It should be noted that restaurants and meals can be votable (by clicking on arrow icons), favorable (by clicking on star icons) and reportable, however there are some exceptions:

- Restaurants that where provided by external APIs can only be favorable, but not votable or reportable;
- Restaurants created by users can be all of them;
- Restaurant meals, either suggested or submitted by users, can be all of them, as a suggested meal associated to the wrong restaurant should be notified, either with reports and/or downvotes.

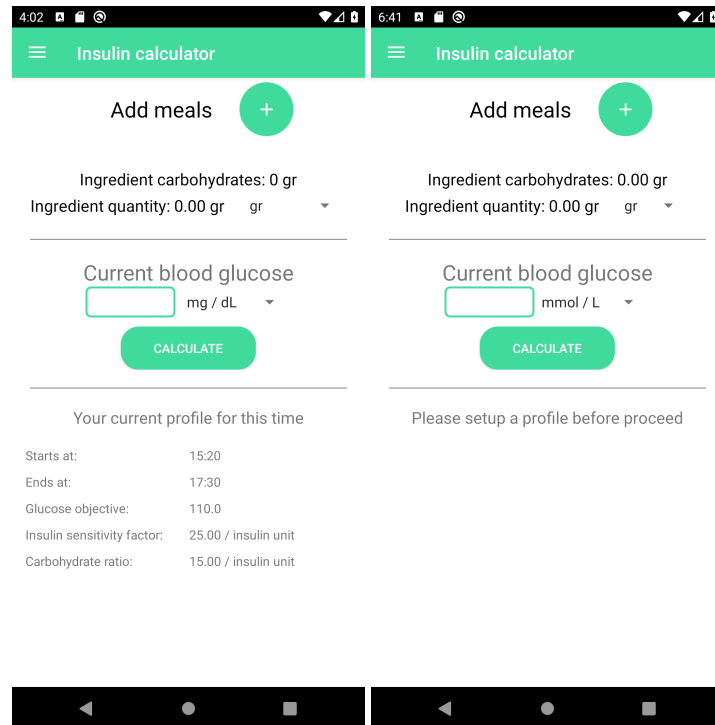
Creating custom meals



As seen above, the user can create custom meals in order to add to a restaurant menu later or to use it with the insulin calculator, which will be explained in the next section.

Using the insulin calculator

To use this feature, the user must create at least one insulin profile which time period matches the current time. If the user does not have a valid insulin profile for the current time, the profile information section in this fragment will appear blank.



(a) Calculator with valid profile (b) Calculator without profile

Having a valid insulin profile, the user must measure its current blood glucose value and select a meal by pressing the plus green button. When this button is pressed, a tab menu will appear where the user can add:

- user's custom meals;

- user's favorite meals

- ingredients from meals;

- suggested meals;

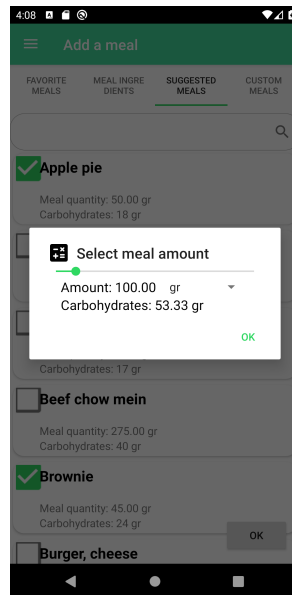
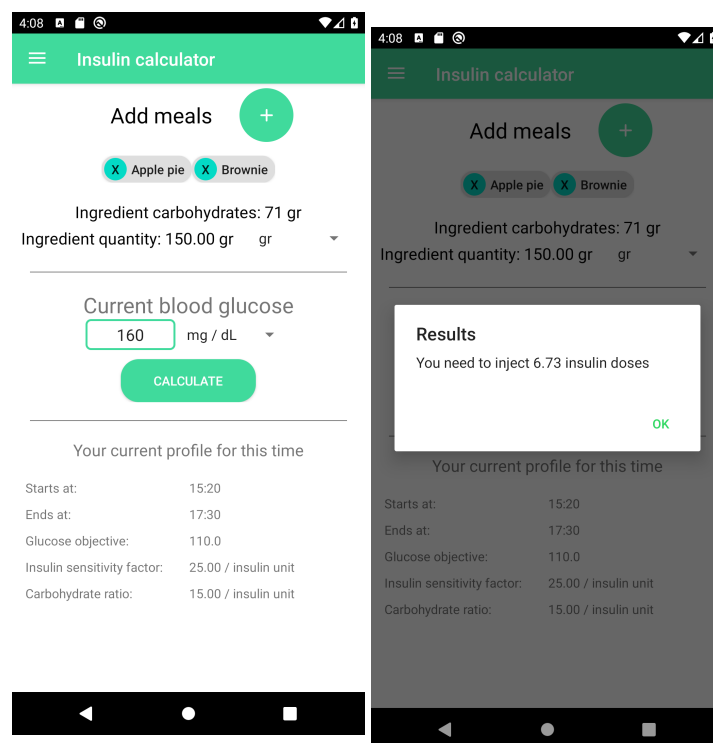


Figure 5.37: Meal selection menu

After these steps, the user is ready to calculate the insulin dosage that corresponds consuming the select meals, with its current blood glucose for its current time.



(a) Calculator with all the valid field filled

(b) Dosage result

5.5 Web browser application

Seeing as the goal of our application is provide nutritional information for meals served in restaurants, we assumed that a typical user would not search for nearby restaurants on their computer's browser.

As such, we decided to develop a minimalist version of our application for browsers with extended moderating functionalities - an authenticated user can only view and create custom meals and edit their insulin profiles; while a moderator can create suggested ingredients and meals for cuisines, and view submission reports and act on them, either by banning the submitter, removing the submission or dismissing the report.

5.5.1 Used technologies

React framework

We chose to build the website with JavaScript [25] using the React framework [26] , as it was the framework lectured in the Web applications development course.

React bootstrap

We chose to use React Bootstrap not only because it offers tools to quickly build responsive and appealing User Interfaces, but also because of some familiarity with it due to previous courses such as Internet Programming.

Additionally, React Bootstrap was chosen over Bootstrap as it is easy to code due to "each component has been built from scratch as a true React component, without unneeded dependencies like jQuery." and "each component is implemented with accessibility in mind. The result is a set of accessible-by-default components, over what is possible from plain Bootstrap." (from <https://react-bootstrap.github.io/>)

5.5.2 Code structure

Single-page application

Regarding design patterns, we decided to develop our web browser application using a Single Page Application approach over a Multi Page Application. This pattern was chosen as it is faster for the user to navigate after the initial load, allows for a better user experience and interaction, and it reduces the load on the server, as most resources (such as HTML and CSS) are only requested and loaded once by the user.

Additionally, React-Router is also used to solve shortcomings that Single Page Applications used to have - the inability to go back and forth on the browser history without reloading the page and the lack of bookmarking.

Routing

Bellow is the diagram containing every possible endpoint that can be visited and bookmarked throughout the application. Green marked endpoints require the user to be authenticated; while red marked ones require the moderator role to access. When this condition fails the user is met with a "You need to be logged in to access this resource" message, or with "You need to be a moderator to access this resource!", respectively.

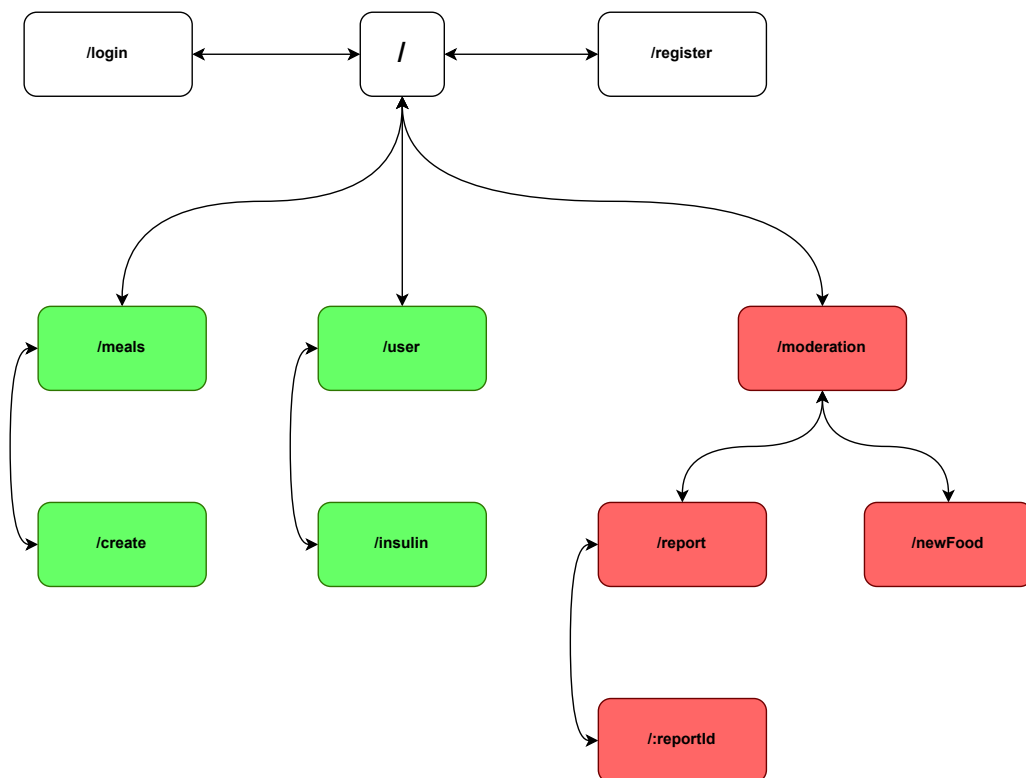


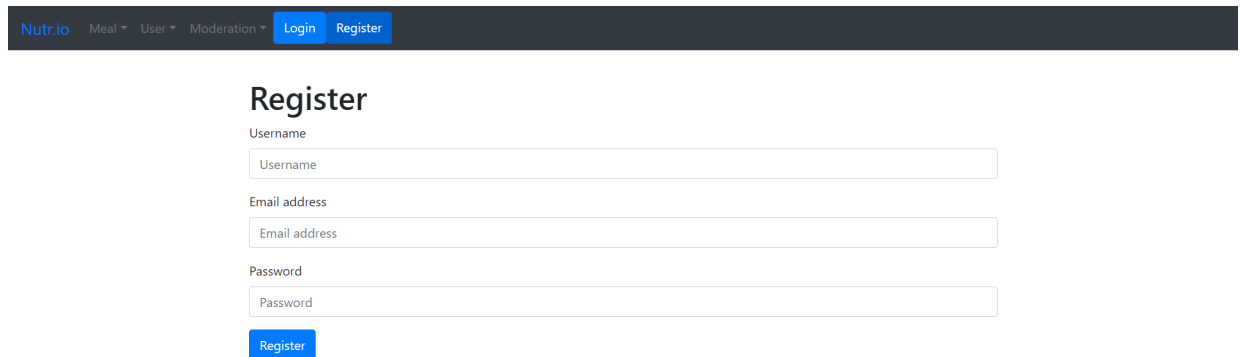
Figure 5.39: The web client's navigation diagram

Error handling

Whenever an HTTP request is sent to the server and an error occurs, the application takes advantage of the **problem-json** content-type to display a specialized error message to the user. When this is not possible, a generic error message is displayed instead.

5.5.3 Functionalities

Register and login



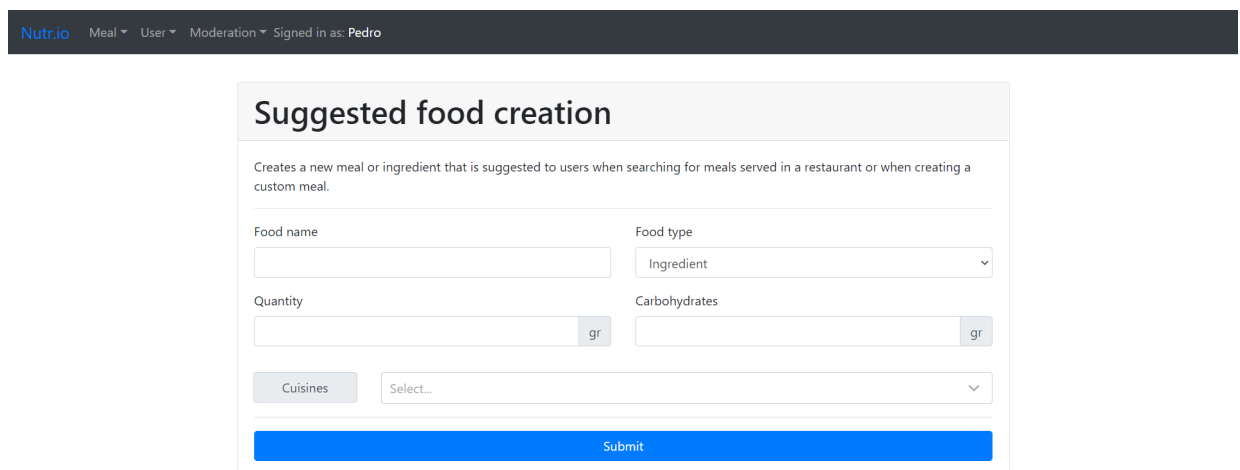
The screenshot shows the top navigation bar of the Nutri.io application. It includes the logo 'Nutri.io' and several menu items: 'Meal', 'User', 'Moderation', 'Login', and 'Register'. The 'Login' and 'Register' buttons are highlighted in blue. Below the navigation bar, the page title 'Register' is displayed. The registration form consists of three input fields: 'Username', 'Email address', and 'Password'. Each field has a placeholder text matching its label. A blue 'Register' button is positioned below the 'Password' field.

Figure 5.40: Register and login page

It should be noted that the landing page will activate more options once the user has logged in.

Moderator: Create a meal or ingredient

This feature provides moderators a way to insert hardcoded meals and ingredients into the database without needing to manually insert them using SQL queries.



The screenshot shows the 'Suggested food creation' form. The form is titled 'Suggested food creation' and has a subtitle: 'Creates a new meal or ingredient that is suggested to users when searching for meals served in a restaurant or when creating a custom meal.' The form contains several input fields: 'Food name', 'Food type' (a dropdown menu with 'Ingredient' selected), 'Quantity' (with a unit selector set to 'gr'), and 'Carbohydrates' (with a unit selector set to 'gr'). There is also a 'Cuisines' dropdown menu with 'Select...' as the current selection. A blue 'Submit' button is located at the bottom of the form.

Figure 5.41: Create a hardcoded meal or ingredient

User: Create a custom meal

Meal creation is done in 5 steps and each can only be advanced when all fields are valid. Going back and forth on each step remembers your choices, so editing a previous input is not cumbersome to the user.

The sum of all ingredient quantities can not be higher than the original quantity. But a meal can always have a leftover quantity which is considered undefined ingredient(s) or ingredients with no carbs.

The screenshot shows the 'Custom meal creation' interface. At the top, a navigation bar includes 'Nutr.io', 'Meal', 'User', 'Moderation', and 'Signed in as: Pedro'. The main heading is 'Custom meal creation'. Below it, a progress bar highlights 'Meal definition'. The instructions state: 'Create a new custom meal linked to your account that can be viewed any time and added to a restaurant on mobile for other users to see.' The form contains three input fields: 'Meal name' with the value 'Fruit salad', 'Quantity' with the value '200' and a unit dropdown set to 'gr', and 'Cuisines' with a dropdown menu showing 'Asian, Austrian'. A blue 'Next' button is at the bottom.

Figure 5.42: Meal description (name, quantity and cuisines)

The screenshot shows the 'Custom meal creation' interface at the 'Ingredients' step. The progress bar highlights 'Ingredients'. The instructions are the same as in the previous step. On the left, there are two tabs: 'Ingredients' (active) and 'Meals'. The main area features a search bar with the text 'No items selected'. Below the search bar, a list of ingredients is displayed, each with a checkbox and a description: 'Beans', '100gr of Beans, broad with 7g of carbs', '30gr of Beans, kidney with 5g of carbs', and '100gr of Lunine bean with 7g of carbs'. A blue button is visible at the bottom left.

Figure 5.43: Ingredients (in the form of basic ingredients and other complex meals) are chosen

Nutr.ioMealUserModerationSigned in as: Pedro

Set the quantity that each component will have:

Banana, with peel

Carbs: 6
Quantity: 40

Apple

Carbs: 1
Quantity: 15

Banana, without peel

Carbs: 16
Quantity: 68

Figure 5.44: User chooses quantity for all chosen ingredients

Nutr.ioMealUserModerationSigned in as: Pedro

Confirm your meal:

Name: Fruit salad

Quantity: 200 grams

Total carbs: 23 grams

Cuisines: Asian, Austrian

Composed by the following ingredients:

40gr of **Banana, with peel** with a total of 6gr of carbohydrates

15gr of **Apple** with a total of 1gr of carbohydrates

68gr of **Banana, without peel** with a total of 16gr of carbohydrates

Previous

Next

Figure 5.45: Input confirmation

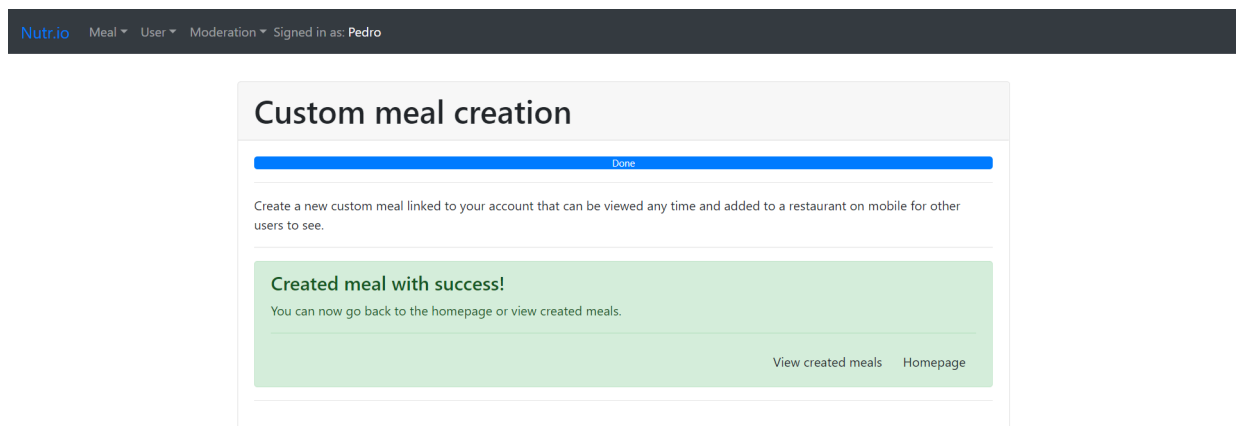


Figure 5.46: Send

User: Create an insulin profile

Like in the mobile application, the user can also create insulin profiles by filling in this form and moving the slider to adjust their time periods.

If certain time periods are already taken by other created insulin profiles, these will appear with gray tone inside the slider.

The screenshot displays the 'Create an insulin profile' form. The top navigation bar is the same as in Figure 5.46. The main content area starts with the text 'You have no profiles! Let's start by creating one.' Below this is a form with four input fields: 'Name (optional)', 'Glucose objective', 'Carbohydrate ratio', and 'Sensitivity factor'. Underneath the form is a horizontal time slider. The slider has a green bar representing the selected time period, which is currently set to the entire 24-hour range. Below the slider, a red bar labeled 'Delete' and a blue bar labeled 'Create' are visible.

Figure 5.47: Insulin profile creation

Moderator: Consult reports

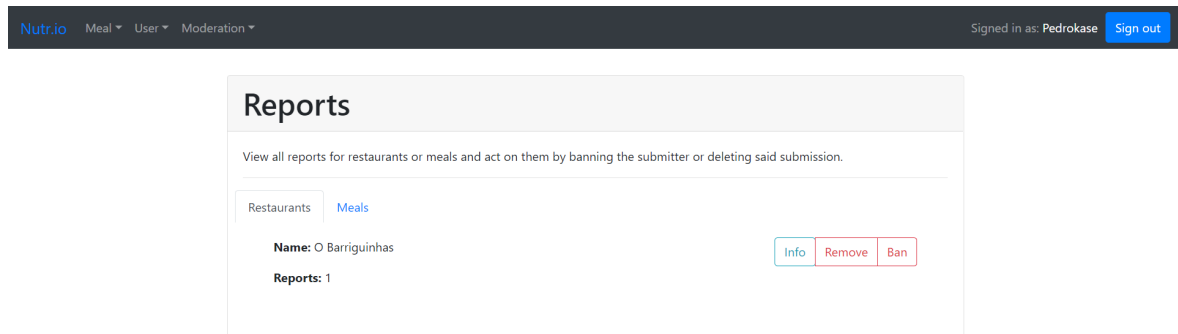


Figure 5.48: List of reports (restaurants and meals)

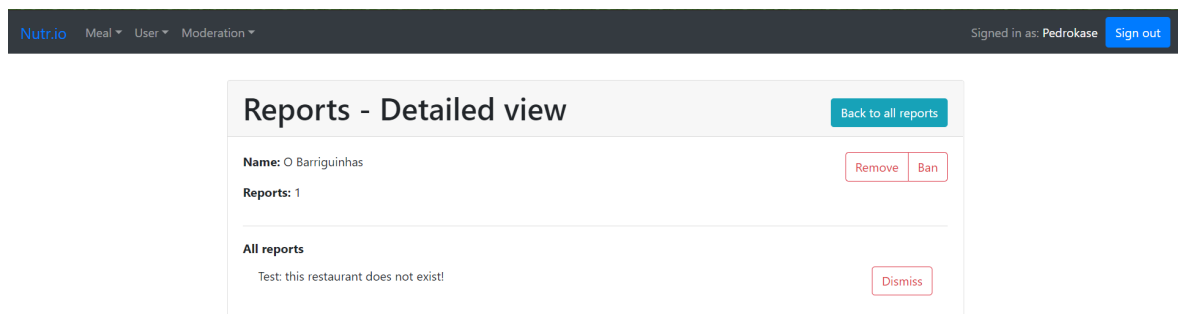


Figure 5.49: A reports detail

Observing a report, a moderator has the options either to remove the faulty submission or ban the user who submitted it.

5.6 Deployment

As planned, we decided to deploy the server to the Heroku[4] cloud platform, to do this we had to follow a procedure and change some settings presented below.

After setting up the CLI tool and creating an app inside Heroku, we started to install the Heroku Postgres plugin which will provide our database to the deployed server.

Configured the plugin, the next step was to create a file called `system.properties` inside the server's root directory with the following setting: `java.runtime.version=11`. This setting forces Heroku to setup the environment with Java JDK 11 when deploying the remote app.

The next step was to create the environment variables, which were being used during development inside the application settings.

As the Heroku application has now our database, the plugin automatically created an environment variable to link the server to the remote tables and, as such, the `spring.datasource.url` value inside the `application.properties` needed to be changed to that variable name, as shown below.

```
spring.datasource.url=${DATABASE_URL}
spring.datasource.username=${PS_POSTGRES_USER}
spring.datasource.password=${PS_POSTGRES_PASSWORD}
spring.datasource.driver-class-name=org.postgresql.Driver
```

Figure 5.50: Environment variable change inside `application.properties`

The last procedure to do before deployment is to run the command `npm run-script build` inside the web client folder, which will execute a script that copies the web client's `index.html` and `main.js` to the HTTP server resources folder, in order to send HTML when an endpoint is being requested for the first time by a client. It is also important to note that the web client is a single-page application, thus only the first request will send HTML + JSON and the consequents only JSON, as the HTML will be cached inside the user's browser.

Concluded those steps, a successful deployment can now be achieved by following the commands inside application's deploy tab.

Use Git to clone `nutrio-app`'s source code to your local machine.

```
$ heroku git:clone -a nutrio-app
$ cd nutrio-app
```

Deploy your changes

Make some changes to the code you just cloned and deploy them to Heroku using Git.

```
$ git add .
$ git commit -am "make it better"
$ git push heroku master
```

Figure 5.51: Heroku deployment procedure

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.

Email verification

Speaking of security, an element required for two-factor authentication would be email verification. This would not be only useful for this feature, as it would provide the user the ability to recover its password, avoid bots from registering in our platform and provide a more safer user account removal, as the actual account removal verifies only the passed JSON Web Token to remove it.

Chapter 7

Appendices

This chapter displays all the appendices referenced in this report.

Appendix A - Database relational model

- **Submitter**

- Attributes: submitterId, creationDate, submitterType
- Primary Key(s): submitterId
- Foreign Key(s): -
- Not null: submitterType

- **User**

- Attributes: submitterId, email, username, password, role, isBanned
- Primary Key(s): submitterId
- Foreign Key(s): submitterId references Submitter(submitterId)
- Not null: username, password, role, isBanned

- **InsulinProfile**

- Attributes: submitterId, profileName, startTime, endTime, glucoseObjective, insulin-SensitivityFactor, carbohydrateRation, modificationDate
- Primary Key(s): submitterId, profileName
- Foreign Key(s): submitterId references Submitter(submitterId)
- Not null: username, password, role, isBanned

- **API**

- Attributes: submitterId, apiName, 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: reportId, submissionId, submitterId, description
 - Primary Key(s): reportId
 - Foreign Key(s): submissionId, submitterId
 - * 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, ownerId
 - Primary Key(s): submissionId
 - Foreign Key(s): submissionId references Submission(submissionId)
 - Not null: restaurantName
- **Cuisine**
 - Attributes: submissionId, cuisineName
 - Primary Key(s): submissionId
- **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, mealType
- Primary Key(s): submissionId
- Foreign Key(s): submissionId references Submission(submissionId)
- Not null: mealName, carbs, quantity

- **RestaurantMeal**

- Attributes: submissionId, restaurantSubmissionId, mealSubmissionId, verified
- Primary Key(s): submissionId, restaurantSubmissionId, mealSubmissionId
- Foreign Key(s):
 - * restaurantSubmissionId references Restaurant(submissionId)
 - * mealSubmissionId references Meal(submissionId)
- Not null: verified

- **Favorite**

- Attributes: submissionId, submitterId
- Primary Key(s): submissionId, submitterId
- Foreign Key(s):
 - * submissionId references Submission(submissionId)
 - * submitterId references Submitter(submitterId)

- **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, cuisineSubmissionId
- Primary Key(s): restaurantSubmissionId, cuisineSubmissionId
- Foreign Key(s):
 - * restaurantSubmissionId references Restaurant(submissionId)

* cuisineSubmissionId references Cuisine(cuisineName)

- **MealCuisine**

- Attributes: mealSubmissionId, cuisineSubmissionId
- Primary Key(s): mealSubmissionId, cuisineSubmissionId
- Foreign Key(s):
 - * mealSubmissionId references Meal(submissionId)
 - * cuisineSubmissionId references Cuisine(cuisineName)

Appendix B - Endpoints' table

This appendix is available inside the folder /docs/appendix in our repository, because of its size.

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] Android crypto. URL <https://developer.android.com/reference/androidx/security/crypto/package-summary>.
- [2] Android fragment. URL <https://developer.android.com/guide/components/fragments>.
- [3] General data protection regulation. URL <https://gdpr-info.eu/>.
- [4] Heroku. URL <https://www.heroku.com/what>.
- [5] International diabetes federation, . URL <https://www.idf.org/aboutdiabetes/type-1-diabetes.html>.
- [6] Worldwide toll of diabetes, . URL <https://www.diabetesatlas.org/en/sections/worldwide-toll-of-diabetes.html>.
- [7] Jdbi. URL <https://jdbi.org/>.
- [8] Android jetpack. URL <https://developer.android.com/jetpack>.
- [9] Android keystore. URL <https://developer.android.com/training/articles/keystore>.
- [10] Ktor. URL <https://ktor.io/>.
- [11] 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.
- [12] Navigation by nav graph view models. URL [https://developer.android.com/reference/kotlin/androidx/navigation/package-summary#\(androidx.fragment.app.Fragment\).navGraphViewModels\(kotlin.Int,%20kotlin.Function0\)](https://developer.android.com/reference/kotlin/androidx/navigation/package-summary#(androidx.fragment.app.Fragment).navGraphViewModels(kotlin.Int,%20kotlin.Function0)).
- [13] Nested navigation in fragments. URL <https://developer.android.com/guide/navigation/navigation-nested-graphs>.
- [14] Project and seminary. URL <https://isel.pt/en/subjects/project-and-seminary-leic>.
- [15] Jdbi result iterable, . URL https://jdbi.org/#_resultiterable.
- [16] Jdbi result iterator, . URL <https://jdbi.org/apidocs/org/jdbi/v3/core/result/ResultIterator.html>.
- [17] Kotlin sequences. URL <https://kotlinlang.org/docs/reference/sequences.html>.

- [18] Android sharedPreferences. URL <https://developer.android.com/reference/android/content/SharedPreferences>.
- [19] Spring's requestattributes, . URL <https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/context/request/RequestAttributes.html>.
- [20] Spring's requestcontextholder, . URL <https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/context/request/RequestContextHolder.html>.
- [21] Spring mvc, . URL <https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html>.
- [22] Java 8 streams. URL <https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>.
- [23] Zomato documentation. URL <https://developers.zomato.com/documentation>.
- [24] P. Corti, T. J. Kraft, S. V. Mather, and B. Park. *PostGIS cookbook*. Packt Publishing Ltd, 2014.
- [25] D. Crockford. *JavaScript: The Good Parts: The Good Parts*. " O'Reilly Media, Inc.", 2008.
- [26] A. Fedosejev. *React. js essentials*. Packt Publishing Ltd, 2015.
- [27] M. Gamboa. *How to Reuse Java Streams*. 2018. URL <https://dzone.com/articles/how-to-replay-java-streams>.
- [28] D. Griffiths and D. Griffiths. *Head First Kotlin: A Brain-friendly Guide*. O'Reilly Media, 2019.
- [29] M. Jones, J. Bradley, and N. Sakimura. JSON Web Token (JWT). RFC 7519, May 2015. URL <https://tools.ietf.org/html/rfc7519>.
- [30] M. Knutson, R. Winch, and P. Mularien. *Spring Security: Secure your web applications, RESTful services, and microservice architectures*. Packt Publishing Ltd, 2017.
- [31] T. Lindholm and F. Yellin. The java tm virtual machine specification, 2nd edn. sun microsystems, 1999.
- [32] P. Oechslin. Making a faster cryptanalytic time-memory trade-off. In D. Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 617–630, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45146-4.
- [33] N. Provos and D. Mazieres. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91, 1999.
- [34] B. Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In R. Anderson, editor, *Fast Software Encryption*, pages 191–204, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. ISBN 978-3-540-48456-1.
- [35] E. Winarno, W. Hadikurniawati, and R. N. Rosso. Location based service for presence system using haversine method. In *2017 International Conference on Innovative and Creative Information Technology (ICITech)*, pages 1–4, 2017.
- [36] H. Zhang, D. She, and Z. Qian. Android root and its providers: A double-edged sword. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1093–1104, 2015.