

Implementation of a Thermal-based Food Recommendation System

Henrique Souza Marcuzzo - 52551

Dissertation presented to the School of Technology and Management of Bragança for the attainment of the Master's Degree in Informatics under the double degree with the Federal University of Technology - Parana

Project developed under the orientation of

Prof. PhD Maria João Varanda

Prof. PhD Juliano Henrique Foleis

Master in Informatics

2022-2023

Implementation of a Thermal-based Food Recommendation System

Project Report

Master in Informatics

Escola Superior de Tecnologia e Gestão

Henrique Souza Marcuzzo - 52551

2022-2023

Dedictory

I humbly dedicate this work to the unwavering love and tireless support of my parents, Rosângela Lopes de Souza and Aguinaldo José Marcuzzo, whose guidance and encouragement have been the foundation of my academic achievements.

In addition, I would like to extend my gratitude to my family for their continued love and support throughout my academic journey. Without their unwavering belief in me, this achievement wouldn't have been possible.

Acknowledgments

I would like to express my sincere gratitude to Prof. PhD Maria João Varanda Pereira and Prof. PhD Juliano Henrique Foleis for their invaluable guidance and unwavering support throughout the course of this work.

Their generous allocation of time, expertise, and resources has been instrumental in shaping the outcome of this research. I am grateful for their mentorship, constructive criticism, and encouragement, which have all been critical in pushing me to achieve my best.

This work was supported by "Fundação La Caixa" and by "Fundação para a Ciência e a Tecnologia" under the scope of the project "Aquae Vitae – Água Termal como Fonte de Vida e Saúde" through the Promove Program ("Projetos I&D Mobilizadores") and also by the Project UIDB/05757/2020 of the Research Centre in Digitalization and Intelligent Robotics. My deepest gratitude goes to these institutions for their essential financial support and belief in our research.

Resumo

O modelo convencional de uma consulta de nutricionista baseia-se na análise dos dados do paciente pelo nutricionista e na provisão de um plano nutricional com base nos objetivos do paciente, e este ciclo é repetido a cada consulta.

Para fazer isso, o nutricionista coleta a maior quantidade de informações possíveis durante a consulta e é com base nesse conjunto de dados que ele estabelece o plano nutricional. Por outro lado, a definição desse plano não é uma tarefa fácil, pois existe uma grande variedade de alimentos e o plano deve seguir muitas restrições e ao mesmo tempo dar alguma liberdade para o paciente escolher entre um conjunto de possibilidades.

Outra questão importante é a possibilidade de monitorar a dieta do paciente, obtendo mais informações para os próximos planos sem a necessidade de agendar uma nova consulta.

Neste projeto, propomos um sistema capaz de coletar dados do usuário para que o nutricionista possa analisar o paciente e fornecer planos nutricionais mais precisos, eficazes e com uma maior variedade de refeições com base nas preferências de cada usuário, tornando a experiência mais agradável para o usuário e auxiliando o trabalho do nutricionista.

Um sistema de recomendação também foi proposto para auxiliar o nutricionista na geração do plano nutricional. Este sistema seria capaz de analisar as preferências do usuário e preencher as refeições do plano nutricional seguindo as regras impostas. Além disso, foi projetado para incluir e priorizar os alimentos termicamente baseados produzidos no contexto do projeto *Aquae Vitae*.

Palavras-chave: Recomendação de Alimentos, Dieta, Inteligência Artificial, Aprendizado de Máquina e Desenvolvimento de Backend

Abstract

The conventional model of a nutritionist consultation is based on the nutritionist analyzing the patient's data and providing a nutritional plan based on the patient's goals, and this cycle is repeated at each consultation.

In order to do that the nutritionist collects the greatest amount of information that it's possible to get during the consultation and is based on that set of data that he establish the nutritional plan. By other hand, the definition of that plan isn't an easy task because there is an huge variety of food and the plan should follow lots of restrictions and at the same time give some freedom to the patient to choose in between a set of possibilities.

Another important issue is the possibility of monitoring the patient diet getting more information for the next plans without the need to schedule a new consultation.

In this project we proposed a system capable of collecting data from the user so that the nutritionist can analyze the patient and provide nutritional plans that are more accurate, effective, and with a greater variety of meals based on each user's preferences, making the experience more pleasant for the user and assisting the nutritionist's work.

A recommendation system was also proposed to aid the nutritionist in generating the nutritional plan. This system would be capable of analyzing the user's preferences and filling in the meals of the nutritional plan following the imposed rules. Moreover, it has been designed to include and prioritize the thermally-based food items produced within the context of the *Aquae Vitae* project.

Keywords: Food Recommendation, Diet, Artificial Intelligence, Machine Learning and Backend development

Contents

1	Introduction	1
2	Problem and goals	3
2.1	Motivation	3
2.2	Overall Objective	3
2.3	Specific Objectives	4
3	Technological Background and State of the Art	5
3.1	Recommendation System	5
3.2	Web Development	9
3.3	Support System for Nutritionists	10
3.4	Conclusion	11
4	Software Modeling	13
4.1	User stories	13
4.1.1	Administrator	13
4.1.2	Nutritionist	14
4.1.3	User	16
4.2	Use cases	17
4.2.1	Administrator	17
4.2.2	Nutritionist	17
4.2.3	User	19

4.3	Entity Relationship Diagram (ERD)	19
4.4	Architecture	22
4.5	Main Flow	22
4.6	Recommendation System	25
4.7	Conclusion	27
5	Backend Development	29
5.1	Overview of Technologies Used	29
5.1.1	Programming Language	30
5.1.2	Database	30
5.1.3	Web Framework	31
5.2	Nutritional Management System Development	31
5.2.1	Organization	32
5.2.2	Module Development	35
5.2.3	Error Handling	43
5.2.4	Pagination	47
5.2.5	Base Entity	51
5.2.6	Documentation	52
5.2.7	Generic Repository	55
5.3	Recommendation System Development	61
5.3.1	Food Ranking	62
5.3.2	Filling Out the Nutritional Plan	66
5.4	Conclusion	69
6	Tests, Evaluation and Discussion	71
6.1	Nutritional Management System Test Development and Results	72
6.1.1	Collected Results	73
6.2	Recommendation System Tests and Results	80
6.3	Conclusion	84

7 Conclusion	87
7.1 Future Works	88

List of Figures

4.1	AquaVitae System Use Cases	18
4.2	AquaVitae Entity Relationship Diagram (ERD).	20
4.3	General architecture of SOA-based systems.	23
4.4	Complete project architecture.	23
4.5	Flow Diagram of the Nutrition Plan Generation.	24
4.6	Recommendation System process overview.	26
5.1	Project Organization <i>aquavitae-app</i>	33
5.2	File <i>config.py</i>	33
5.3	Directory <i>aquavitae-app/src</i>	35
5.4	File <i>src/main.py</i>	36
5.5	Directory <i>src/modules/domain/food</i>	37
5.6	Directory <i>src/modules/domain/food/entities</i>	37
5.7	Part 01: File <i>src/modules/domain/food/entities/food_entity.py</i>	38
5.8	Part 02: File <i>src/modules/domain/food/entities/food_entity.py</i>	39
5.9	File <i>src/modules/domain/food/repositories/food_repository.py</i>	39
5.10	File <i>src/modules/domain/food/services/food_service.py</i>	40
5.11	File <i>src/modules/domain/food/controllers/food_controller.py</i>	41
5.12	Directory <i>src/modules/domain/food/dto/food</i>	42
5.13	File <i>src/modules/domain/food/dto/food/food_dto.py</i>	43
5.14	File <i>src/modules/domain/food/dto/food/update_food_food_dto.py</i>	43
5.15	Example of handling errors extracted from the <i>FastAPI</i> documentation.	44

5.16	File <i>src/core/common/dto/exception_response_dto.py</i>	45
5.17	File <i>src/core/types/exception_types.py</i>	45
5.18	Part 1: File <i>src/core/types/http_exception_handler.py</i>	46
5.19	Example error message extracted from the FastAPI documentation.	47
5.20	Part 2: File <i>src/core/handlers/http_exceptions_handler.py</i>	48
5.21	File <i>src/core/common/custom_error_response.py</i>	49
5.22	File <i>src/core/decorators/pagination_decorator.py</i>	50
5.23	File <i>src/modules/infrastructure/database/base_entity.py</i>	53
5.24	File <i>src/core/common/dto/base_dto.py</i>	53
5.25	Create Food request body.	54
5.26	Create Food response body.	55
5.27	Variables of an endpoint with pagination.	56
5.28	Get Food detailed response body with pagination.	57
5.29	Get Food response body with pagination in <i>JSON</i> format.	58
5.30	Error response body in <i>JSON</i> format.	59
5.31	File <i>src/modules/infrastructure/database/soft_delete_filter.py</i>	60
5.32	Method <code>__apply_options</code> inside the file <i>.../database/base_repository.py</i> . .	61
5.33	File <i>.../recommendation_system/controllers/recommendation_system_controller.py</i>	62
5.34	File <i>.../interfaces/find_user_food_preferences_interface.py</i>	63
5.35	File <i>.../interfaces/find_user_food_preferences_interface.py</i>	64
5.36	File <i>.../interfaces/complete_nutritional_plan_interface.py</i>	68
5.37	File <i>.../interfaces/complete_nutritional_plan_interface.py</i>	69
6.1	File <i>test/fixtures/food.json</i>	73
6.2	File <i>test/utils/database_config_test_utils.py</i>	74
6.3	File <i>conf/test.py</i>	75
6.4	File <i>test/test_base_e2e.py</i>	76
6.5	File <i>src/modules/domain/food/tests/test_food_e2e.py</i>	77
6.6	File <i>src/modules/domain/food/tests/test_food_e2e.py</i>	78

6.7	Test success rate	78
6.8	Code coverage report from <i>pytest-cov</i>	79
6.9	Response time distribution of the system endpoints	79
6.10	Food test response times	80
6.11	<i>Base Repository</i> create method	80
6.12	Nutritional plan without any input	82
6.13	Nutritional plan with historical consumption data only	83
6.14	Nutritional plan with user preferences only	83
6.15	Nutritional plan with user preferences and consumption history	84

Acronyms

API Application Programming Interface. 3, 4

CPS Cyber Physical Systems. 9

CRUD (Create, Read, Update, Delete). 85

DTO Data Transfer Object. 34, 38, 39, 41, 42, 52

ERD Entity Relationship Diagram. xiv, 13, 20

ORM Object-Relational Mapping. 4, 30, 35, 37

RS recommendation systems. 2, 4–10, 13, 25–27, 30, 84, 87, 88

SOA Service-Oriented Architecture. 22, 27, 31, 32

STARec Search-based Time-Aware Recommendation. 8

Chapter 1

Introduction

People seek nutritionists for a variety of reasons ranging from helping them lose weight and/or gain muscle mass to changing their eating habits to a healthier diet that improves their quality of life. And according to some [1] studies, a good diet has many benefits for the patient, such as helping to prevent disease.

However, in some offices there is still a barrier between the patient and the nutritionist, which consists of a consultation with the nutritionist from time to time, the patient receiving a nutritional plan, and after the established time of this plan a return to the office to observe the patient's progress.

In this model the contact between both parties is minimal. On the nutritionist's side, it doesn't allow him to have enough data to evaluate if the proposed nutritional plan was adequate or not, if the patient is following the plan and how to calibrate it to get better results in future sessions. Moreover the nutricionist should have an easy and efficient way of defining the plan having the help of an intelligent system that shows him the food options making automaticaaly the calculations and suggesting several possibilities for each meal.

And on the patient's side, the actual system doesn't allow him to quickly and reliably have control over what food was consumed during the period proposed by the nutritional plan, control over the amount of calories ingested, and also generates the concern of what he should eat at the next meal and if he has all the necessary items for the next meal

according to the nutritional plan.

Aquavitae’s project proposes, not only to include thermal-based food items but also as an initial idea, to help both parties overcome these challenges. This allows the user to have quick access to their nutritional plan, to mark the meals they have consumed during the day, and to indicate which foods have restrictions or preferences so that the system can adapt to the user’s profile. In continued consideration of the patient, the system also aims to provide meal recommendations that respect the nutritional plan, thereby allowing for a greater diversity of meals.

And for the nutritionist, the system will be able to generate detailed reports about the analyzed patient, so that with this information the nutritionist will be able to generate more accurate nutritional plans that bring more results for the patient.

Considering the recommendation systems (RS), which is anticipated to add artificial intelligence features as a differential to the application, it should be able to identify the restrictions either self-imposed by the user or recommended by the nutritionist, as well as the patient’s preferences. This system aims to fill the nutritional plan accordingly, generating varied meals to prevent the patient from becoming weary of consuming the same items repeatedly, thus enhancing the user experience.

The RS must also ensure compliance with other rules of the nutritional plan not previously mentioned, such as calorie limits. Moreover, the system is designed to prioritize the inclusion of high-priority foods, particularly thermally-based food items, within the nutritional plan.

For easier navigation, this document is organized as follows: Chapter 2 details the motivations leading to the development of this work. Chapter 3 contains the research conducted on different techniques and challenges related to RS. The backend modeling and the RS are presented in Chapter 4. Important points of the development process are discussed in Chapter 5, while Chapter 6 elaborates on the tests conducted for the project. Lastly, the conclusion of this work is presented in Chapter 7.

Chapter 2

Problem and goals

2.1 Motivation

Nutritionists required an application to streamline the management of patient information, simplify the creation of nutritional plans, and closely monitor their patients' diets to ensure adherence to the prescribed nutritional plans. This system would facilitate the tracking of each patient's progress, or lack thereof, during return visits to the consulting room. Furthermore, it would collect valuable data to inform the development of subsequent plans tailored to the patient's profile.

It was also of interest that the system could somehow suggest other types of meals that could match the nutritionist's recommendations, making the nutritional plan more enjoyable for the patient who would not be stuck to a restricted diet and could vary the meals without compromising the nutritional plan objective.

2.2 Overall Objective

The main goal of this work is to design and implement an API capable of storing and managing the information provided by the nutritionists and users, turning easier the task of defining the meals making automatic calculations of the nutritional parameters, Moreover the application should also be capable of recommending meals to the users as

an alternative to what is suggested by the nutritionist but maintaining the nutritional plan, aiming to make the nutritional plan less stressful to the patient. Moreover, the food table was enriched with the thermal-based items produced under the scope of the project *Aquae Vitae*.

2.3 Specific Objectives

In addition to building the API to facilitate the work of the nutritionists and improve the user experience with the meal RS, there is also the goal and challenge of learning the technologies used to build the Backend, and the RS, which are:

- **Backend:**

- FastAPI, the framework used to build the API, the server that will handle the requests made by the Web or Mobile application;
- PostgreSQL, for data storage;
- SQLAlchemy, a powerful Object-Relational Mapping (ORM) used by many other Python based frameworks to make it easier to manipulate the data in the database;
- Pydantic, to help with OpenAPI documentation and to ensure that the data received and returned in a request is in the expected format;
- Pytest, to test the system's endpoints and methods ensuring that changes made to the system don't impact the service in an unwanted way.

- **Recommendation system:**

- Study what strategies are used in the market today and identify the best approach to use that meets the system requirements;
- Study and enhance existing knowledge of python artificial intelligence and machine learning.

Chapter 3

Technological Background and State of the Art

This chapter will discuss the state of the art for the different types of RS that have been adopted by the industry, and then present and justify the technologies used for the development of the backend application and also the RS, that are part of the project.

3.1 Recommendation System

In an overview there are different approaches to develop RS and the article [2] gives us a brief summary for these different approaches, as well as advantages and disadvantages, which contexts to use a certain algorithm and so on. To complete these categorizations the article [3] encompasses different approaches into three major categories which are global models, personalized models and also hybrids models that are a combination of these two, taking advantage of the characteristics of both. A brief explanation about each category according to the same article is:

- **Global models:** It requires a large amount of data from several users being transmitted to the server to process and retrain the model on the server side. This type of approach is excellent when the system doesn't have much data from one user

and starts to make recommendations based on the majority preferences. This ends up distorting the recommendations of users who are already enrolled in the system and providing information to him for a longer period of time. They may end up receiving recommendations based on mass and not reflecting their preferences. The paper [4] provides us with a great example of building a framework based on this kind of approach, and [5] has done a study of session-based recommendation systems using this model, making comparison even with linear recommendation models, and proposed a method that proved superior to competitors in the same segment present in the state of the art.

- **Custom models:** Can be trained and used on the client side itself, so they can be used off-line as they don't require data to be transmitted to the server. This approach provides recommendations based entirely on the user's preferences, but has the disadvantage of the risk of not having enough data for more accurate recommendations.
- **Hybrid models:** Tries to balance the two learning models already described, where it seeks to transfer the knowledge of global preferences to the user's personal preferences in order to make personalized recommendations without leaving out popular items in the system. A good article for better understanding this model is [6] which has done a study on this kind of approach in RS by even proposing an algorithm that uses it.

There are also options for recommendations based on learn-to-rank algorithms ([7], [8] and [9]), which consist of learning what the user is most likely to consume based on past interactions, which also fit into previously defined *custom models*. While the article [7], suggested a new approach for this type of learning that has proven to be superior to other pointwise and pairwise algorithms, [8] preferred to compare known algorithms and use the one with the best performance for the problem explored, in this case the *RankSvm* algorithm was the chosen one.

RS based on continuous learning suffer from the catastrophic forgetting problem as pointed out by the article [10], where the RS when updating the cycle of a training base, ends up forgetting past interactions, so the article in question proposes an algorithm that carefully selects which interactions will be chosen to update a new cycle. Although the article has obtained better results than the ones analyzed, as the author himself says, interactions of past items may be recurrent in future interactions and therefore user preferences are preserved.

The article [11] suggests a collaborative approach for RS where it would use data from neighbors who have similar preferences to compose the suggested items. That is something similar to what is proposed by the article [3] in *hybrid models*, because as it was also pointed out by the article, systems that are based only on the user have the same problems pointed out in *custom models*.

If the dataset contains missing attributes it is possible, in some cases, to fill them with information from other domains as detailed in the article [12] using the technique known as cross-domain. Another interesting point highlighted by the article was the amount of neurons used in the deep learning model that produced influences the hit rate, where there is a global optimum of the amount of neurons in a single layer, after that, increasing the amount of neurons will only make the system less efficient.

To better understand some aspects of deep learning and how to use it in RS as well as techniques for validation and also cross domain as mentioned in the paragraph above the article [13] and [14] is recommended, as it provides a good analysis of the topic.

The paper [15], developed a framework based on meta-learning that meets all the characteristics cited in *hybrid models*, such as catastrophic forgetting and cold start. In addition the framework is able to adapt quickly to the entry of new users providing new suggestions in a short period of time. Proving its efficiency when compared to other algorithms in the same segment of RS.

Comparing the effect that implicit and explicit data collection for the RS has on the user experience is critically important to the success of the application and was very well scored by [16] and [17].

The work done by [16] explored this topic with machine learning techniques ranging from supervised matrix factorization, contextual bandit learning to Q learning and concluded that each type of approach produces different effects. However Bandit-Two and Bandit-Four approaches are proven superior.

While [17] explored the advantages and disadvantages of the *exploration* and *exploitation* approaches, it's crucial to understand these two concepts better. The *exploration* approach pertains to exploring a diverse set of possible options without prioritizing what is already known to work well. This method encourages diversity and novelty but can potentially introduce more risk and uncertainty. On the other hand, the *exploitation* approach focuses on using known information or solutions that have previously proven effective, minimizing risk but potentially limiting diversity and novelty. The goal is to balance these two approaches to generate a method that continually enriches the system's knowledge over time without causing undue stress to the user.

The work done by [18] points out that depending on the type of application, it becomes interesting to observe the micro behaviors performed by the user to create the recommendations that will be displayed to him in a next session. This article also developed a new algorithm *MKM-SR*, which is a combination of algorithms *M-SR* and *KM-SR*, and with the proposed algorithm got better performances than those present in the state of the art until its publication.

The work [19] as well as the above paper observe other types of user behavior mixing with past preferences to predict user demands over time. It was developed the Search-based Time-Aware Recommendation (STARec) algorithm which compared to other algorithms (such as *LSTM*) subjected to the same tests had a better performance.

Another problem that happens frequently in RS is working with very dense databases, that is, with many attributes and many items to be calculated, which decreases the performance of the application, for this there are techniques known as *Embedding* that reduce the dimension of the data and the article [20] give us a brief explanation about the topic.

Another approach to build a RS is using decision tree algorithms as shown in the

paper [21], which compares the processing time and presents the challenges faced by this approach. The paper also proposes an approach that has logarithmic scale complexity and is shown to be superior to other comparative approaches present in the state of the art.

There are rule-based RS approaches, and to better understand how these work the articles [22] and [23] are of interest. And [23] also explains the differences in metrics for evaluating a system and when to use them.

There are also strategies for RS based on fuzzy logic, a great study by [24] compared algorithms of if-then rules with algorithms of decision tables and came to the conclusion that systems that use if-then rules get better performance. And the work done by [25] showed that increasing the amount of if-then rules does not always increase the effectiveness of algorithms, and there is an overall complexity optimum for the problems studied.

The work done by [26] proposes to generate if-then rules algorithms automatically for numerical data, and is a great case study for solving similar problems.

Fuzzy if-then rules algorithms have also been used in Cyber Physical Systems (CPS) in the medical and hospital sector in order to reduce false alarms generated by the sensors, increasing their efficiency and reducing costs and resources, as shown in the work done by [27].

3.2 Web Development

Regarding technologies for web services the research conducted by [28] about the RESTful protocol demonstrates several points and benefits of using it in the developed projects in comparison with other development patterns. And the [29] article, although not focused on studying the REST protocol for web development, which has been studied for decades, provides a great comparison of this standard and comparisons with other protocols that demonstrate the superiority of REST for web service development.

Along with the REST protocol, another work by [30] conducts a study on the architecture of microservices and the impacts this development pattern brings to a project. It highlights the benefits and challenges of using it. Benefits were noted as independent deployment, ease of scaling the applications, maintainability, and no commitment to a single technology stack. The challenges were identified as complex distributed transactions, testing the entire system, and dealing with service faults.

From this survey, it was possible to analyze several different techniques for developing Recommendation Systems (RS) that use a range from Machine Learning, Learn to Rank, Decision Trees, Global/Custom/Hybrid models to Fuzzy logic with if-then rules.

Given the current formulation of the project, we believe that the fuzzy logic technique is the most appropriate for the project since there are a number of pre-defined rules for the recommendation of a food or meal to the user. More explanations will be given in the modeling chapter.

We also concluded that using the RESTful protocol for communication and a microservices-based architecture would be most appropriate for the project because of the benefits it brings to the work to be developed, as seen in the articles related to this work.

3.3 Support System for Nutritionists

When it comes to support systems for nutritionists the [31] proved to be interesting because it has a certain similarity to the results we seek to achieve even though the technique used is different.

The work [31] mentioned above was done in Chile, where there was a high rate of deaths from cancer and studies showed that 40% of the types of cancer are linked to obesity, another problem that was present in at least 35% of the still young population of the country. So the work focused on consulting several nutritionists in order to generate automatic plans for patients based on the problem that the patient should face and built a software for this based on the rules defined by the nutritionists consulted.

Another interesting work was done by [32] who built an app for patients to stay healthy

in times of pandemic with nutrition plans from nutritionists and with goals to be met that could be tracked frequently by the mobile app they developed, and the researchers found that this kept the patients motivated to continue and complete the nutrition plan.

Therefore, it's possible to observe that technological solutions are emerging for Web (eHealth) and Mobile (mHealth) services, and tend to become more frequent in the daily lives of patients and health professionals. And they also bring benefits that encourage patients to follow the proposed nutritional plans more vigorously.

3.4 Conclusion

In light of the comprehensive literature review and assessment of available technologies, we have initiated the back-end modeling phase. This approach was informed by our knowledge of the web technologies to be used and the most appropriate artificial intelligence techniques for implementing the recommendation system. This thorough grounding in relevant literature and technological capabilities forms the foundation for the next steps of our project. As we move forward, we expect to continuously refine and optimize our approach in order to deliver a highly effective and user-friendly solution.

Chapter 4

Software Modeling

In this chapter we will cover the key points of the backend modeling of the software developed which includes all the requirements such as user stories, use cases and also the ERD of the database and the architecture used for development. We will also detail how we modeled the RS and the steps it will follow.

4.1 User stories

For a better understanding of the user stories we will separate them by the actors that are present in the system, which are *Administrator*, *Nutritionist* and *User*.

4.1.1 Administrator

The administrator is able to manage all the system's internal data in order to facilitate the work that will be done by the nutritionist, which are:

1. The administrator is able to manage the role that each user has within the system in order to limit their access within the system for security purposes;
2. The administrator is able to manage all the internal data in the system that will support the nutritionist in his role;

- (a) As the administrator, I want to manage the goals that an appointment can have;
- (b) As the administrator, I want to manage the types of pathology that a user can be diagnosed with;
- (c) As the administrator, I want to manage the activity levels a user can have;
- (d) As the administrator, I want to manage the types of specificity;
- (e) As the administrator, I want to manage the food categories;
- (f) As the administrator, I want to manage the food table;
- (g) As the administrator, I want to manage the meal types;
- (h) As the administrator, I want to manage which foods can be consumed in a certain type of meal.

4.1.2 Nutritionist

The nutritionist is able to perform a set of actions that will allow to define easily and better the nutritional plan. The most important actions are:

1. As the nutritionist, I want to register users in the system;
2. As the nutritionist, I want to have access to the user's personal data;
3. As the nutritionist, I want to schedule appointments for the users;
 - (a) As the nutritionist, I want to specify what the goals are for a particular scheduled appointment.
4. As the nutritionist, I want to manage the anthropometric data of the users;
5. As the nutritionist, I want to manage the biochemical data of the users;
6. As the nutritionist, I want to manage the diagnoses of each user;

7. As the nutritionist, I wish to manage the pathologies of each user;
8. As the nutritionist, I wish to add foods that a user cannot eat under any circumstances;
9. As the nutritionist, I want to create nutritional plans for the users;
10. As the nutritionist, I want to specify the limit of calories, lipids, proteins and carbohydrates for each nutritional plan;
11. As the nutritionist, I wish to specify that the limits imposed in the nutritional plan will be daily, weekly, or per meal;
12. As the nutritionist, I want to manage forbidden foods for a specific nutritional plan;
13. As the nutritionist, I want to manage the amount of meals that a nutritional plan will have;
 - (a) Specifying the timing of each meal in a nutritional plan;
 - (b) Specifying the types of each meal in a nutritional plan.
14. As the nutritionist, I want to have the option to add multiple food options to a single meal so that the user has options to choose to avoid getting tired of the nutritional plan;
15. As the nutritionist, I wish to be unable to add foods or meals to the user's nutritional plan that have items previously defined as prohibited;
16. As the nutritionist, I want to receive alerts when tries to add foods to the nutritional plan that exceed the previously defined limits to be consumed in order to decide whether or not to keep the food in question;
17. As the nutritionist, I want, after filling out an entire day of the user's nutrition plan, to be able to ask the system to fill out the rest following the rules defined earlier;

18. As the nutritionist, I wish to validate and change if necessary the meals automatically filled in by the system before finishing the user's nutritional plan;
19. As the nutritionist, I wish to create meal options with various foods;
 - (a) Specifying the amount to be consumed of each food for each meal created.
20. As the nutritionist, I want to have access to the user's diary in order to observe how the user's consumption pattern has been;
21. As a nutritionist, I want to see the nutritional plan of the patients from time to time to take as a basis for schematizing new nutritional plans.

4.1.3 User

The user is able to perform actions to keep his data updated and inform the system which foods he has been consuming, which will help the nutritionist in the next appointments. The most important actions are:

1. As the user, I want to have access to my nutritional plan in order to visualize what I should consume during the time determined by the nutritionist;
2. As the user, I want to be able to select which foods I have consumed in a particular meal;
3. As the user, I want to be able to report foods that I have consumed that weren't in the nutritional plan;
4. As the user, I wish to have access to my food diary in order to have control over my consumption pattern;
5. As the user, I want to have statistics on how the nutritional plan is going, in order to see if the proposed goals are being achieved or not;
6. As the user, I want to be able to manage foods that I have a preference for;

7. As the user, I want to be able to manage foods for which I have no preference;
8. As the user, I want to be able to manage my personal data in order to always keep it up to date;
9. As the user, I want to visualize my appointments;
10. As the user, I wish to receive notifications of upcoming appointments;
11. As the user, I want to view my diagnostics;
12. As the user, I want to visualize my pathologies.

4.2 Use cases

In this section we will explore how the three actors mentioned above will interact with the system through the use case diagram that provides a better understanding of how the system will work as a whole, as can be seen in the Figure 4.1.

As can be seen in the Figure 4.1 above, some of the use cases are representing more than one user story in a single element.

To better explain this we will point out which user stories are being represented by each use case, for this we will separate again by system actors.

4.2.1 Administrator

- *Manage internal system data*: User Stories 2, including 2a to 2h;
- *Manage user role*: User story 1.

4.2.2 Nutritionist

- *Register user*: User Story 1;
- *Manage appointments*: User Stories 3 and 3a;

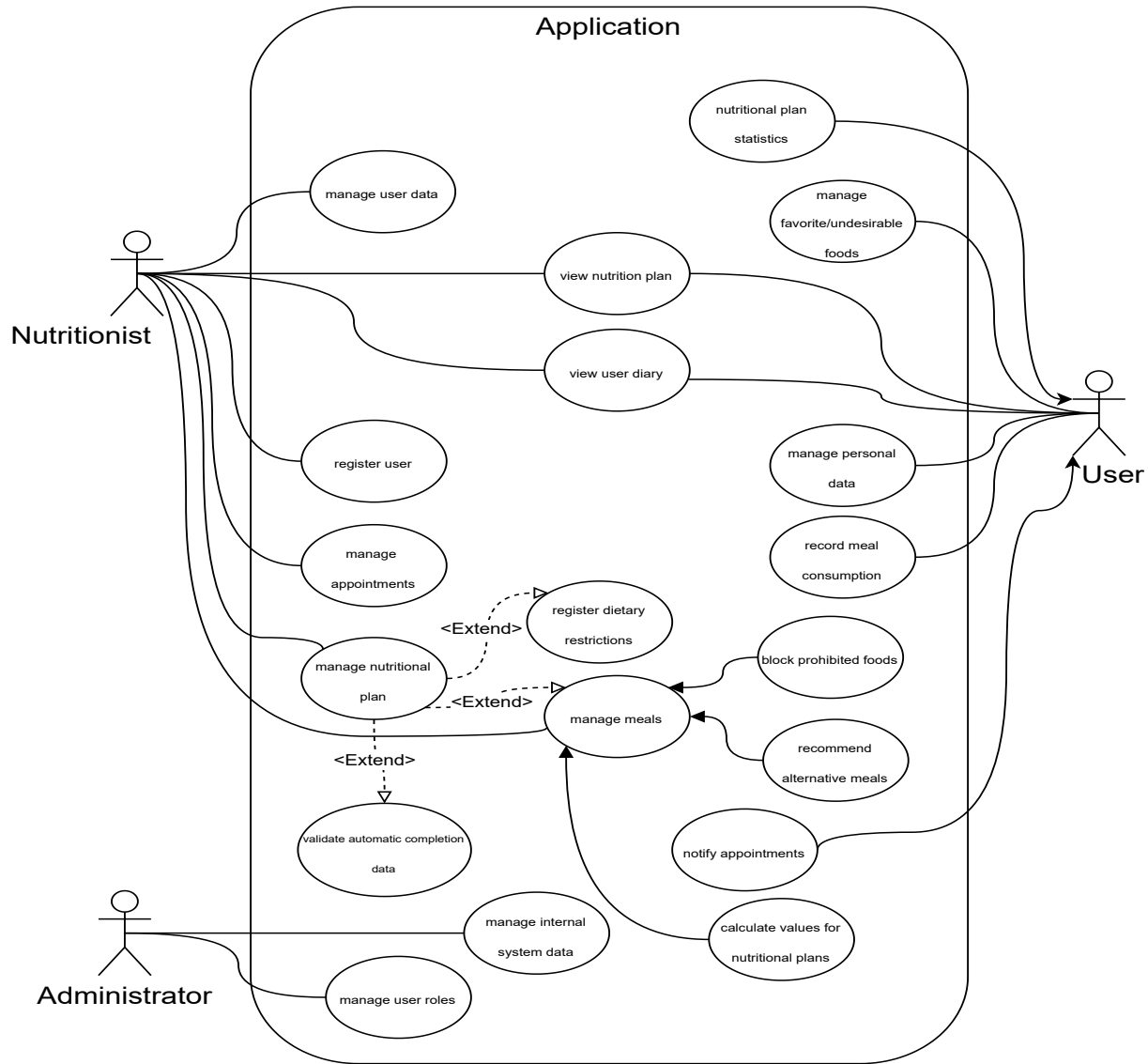


Figure 4.1: AquaVitae System Use Cases

- *Manage nutritional plan*: User Stories 9, 13, 13a and 13b;
 - *Register dietary restrictions*: User Stories 10 and 11;
 - *Manage meals* User Stories 14, 15, 19 and 19a:
 - * *Block prohibited foods*: User Story 12;
 - * *Recommend alternative meals*: User Story 17;
 - * *Calculate values for nutritional plans*: User Story 16;

- *Validate automatic completion data*: User Story 18;
- *Manage user data*: User Stories 2, 4 to 8;
- *View user diary*: User Story 20;
- *View nutrition plan*: User Story 21.

4.2.3 User

- *Nutritional plan statistics*: User Story 5
- *Manage favorite/undesirable foods*: User Stories 6 and 7
- *View nutrition plan*: User Story 1
- *View user diary*: User Story 4
- *Manage personal data*: User Stories 8, 11 and 12
- *Record meal consumption*: User Stories 2 and 3
- *Notify appointments* User Stories 9 and 10

4.3 Entity Relationship Diagram (ERD)

After gathering all the requirements we moved on to the bank modeling phase so that it could represent and support all the user stories in a structured and organized way without duplicating data and maintaining the data coherence. You can see the diagram through the Figure 4.2 below.

Some points that are of importance to understand some decisions made are:

- The table *antecedent* represents the pathologies of the users;

- The *specificity* table has the function of storing the foods that the user has preferences or not, or even foods for which the user has some type of allergy. The type that each food in the table represents is defined by the secondary table *specificity_type*;
- The table *food* has its base values fixed in a quantity of 100 grams, so there was the need to create the table *item* where each element of *food* would have at least one element in *item* to be able to specify the quantity in grams;
 - With the explanation of the item above there would be no need for the table *item_has_food*, however the table *item* has a second function that is to compose meals with more than 1 food with different quantities to be consumed and that is the reason why there was the need for the separation giving rise to the table *item_has_food*.
- According to the requirements of the nutritionist, 13, 13a and 13b there should be specific types of meals where the amount of nutrients to be consumed would be predefined, however the time that each user would make this meal would be variable according to the routine of each one and therefore specific to each nutritional plan, giving then origin to the tables *type_of_meal* and *meals_of_meal* respectively;
 - By establishing fixed meal types system-wide via the *type_of_meal* table, the system can determine which foods are permissible or ideally consumed during a particular meal type. This led to the creation of the intermediary *food_can_eat* table.
- Since a meal can have several food options to be consumed there was a need for the *meals_options* table, however, since the user can report consuming items that were not in the nutritional plan for that meal it was necessary to create the *diary* table for this.

4.4 Architecture

It was decided to use a Service-Oriented Architecture (SOA) for the development of this API, with structures similar to the NestJS [33] framework. The development team had prior experience with this technology, which would make the transition and adaptation of the technologies used for this project easier.

Before demonstrating how the project was structured, it's important to define what SOA is. According to IBM [34], *"SOA, defines a way to make software components reusable and interoperable via service interfaces. Services use common interface standards and an architectural pattern so they can be rapidly incorporated into new applications. This removes tasks from the application developer who previously redeveloped or duplicated existing functionality or had to know how to connect or provide interoperability with existing functions."*

It's worth noting that while SOA shares some similarities with microservices architecture, there are important differences between the two. While microservices are a specific implementation of SOA, they are more focused on breaking down large monolithic applications into smaller, independent services. In contrast, SOA is a more abstract concept that encompasses a broader range of approaches to building distributed systems.

With this definition it is possible to see in the Figure 4.3 below how the project was structured and what path each request will take until it gets a response.

It is important to note that this structure is applied to each module individually and does not represent the system as a whole that could be represented as in the Figure 4.4 where each module has its *Controller*, *Service* and *Repository* layer and works independently of the entire system.

4.5 Main Flow

Finally, a brief explanation about the central activity of the system to be developed, which is the generation of nutritional plans for the patients as well as the user registering what

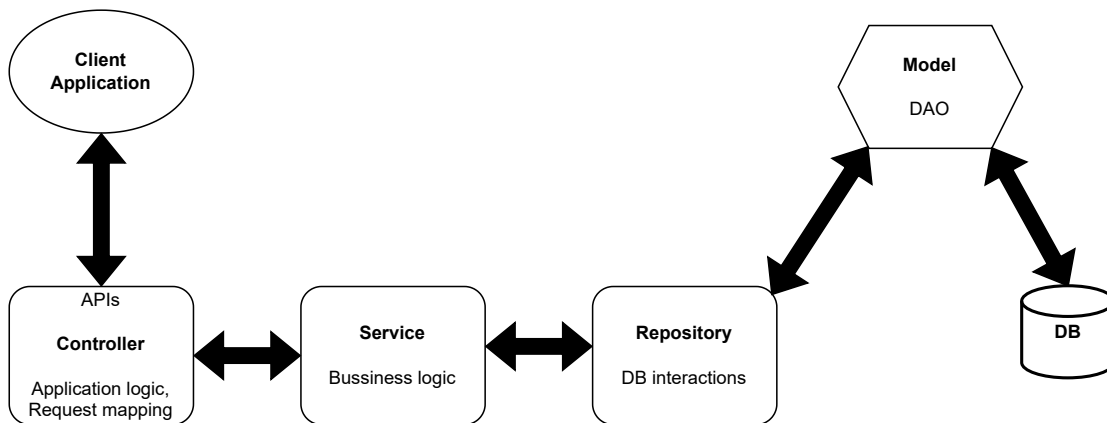


Figure 4.3: General architecture of SOA-based systems.

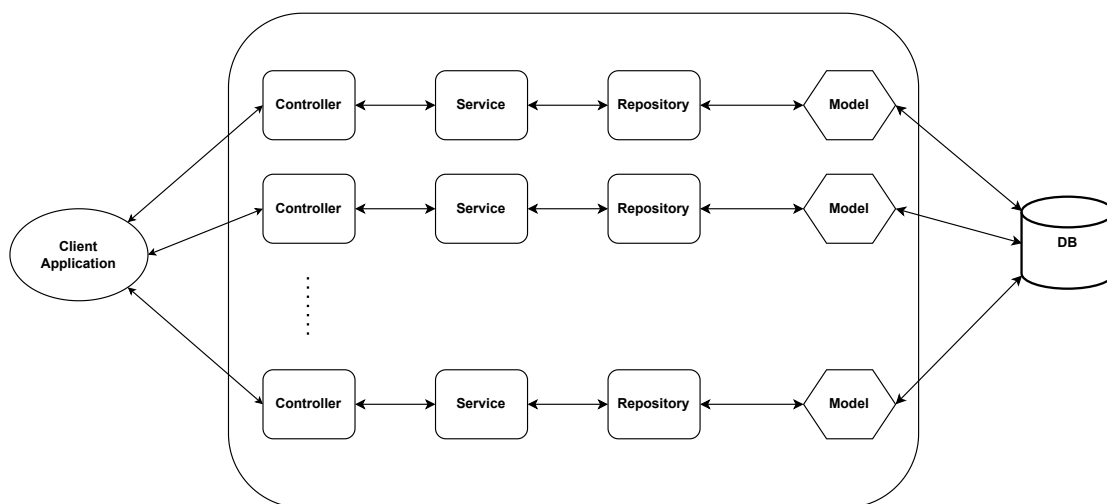


Figure 4.4: Complete project architecture.

is being consumed.

It is possible to observe this flow by the Figure 4.5. Where we can observe in the following order:

1. The nutritionist generates the nutritional plan rules based on the user's personal data;
2. With the specified rules it is possible to filter the foods, leaving available to include in the nutritional plan only those that the patient can consume;
3. The Nutritionist validates that the foods he includes are in accordance with the plan's proposal and meet all the requirements;
4. After approval the data is stored in the bank and passed to the user;
5. The user starts the nutritional plan and informs the system which foods are being consumed.

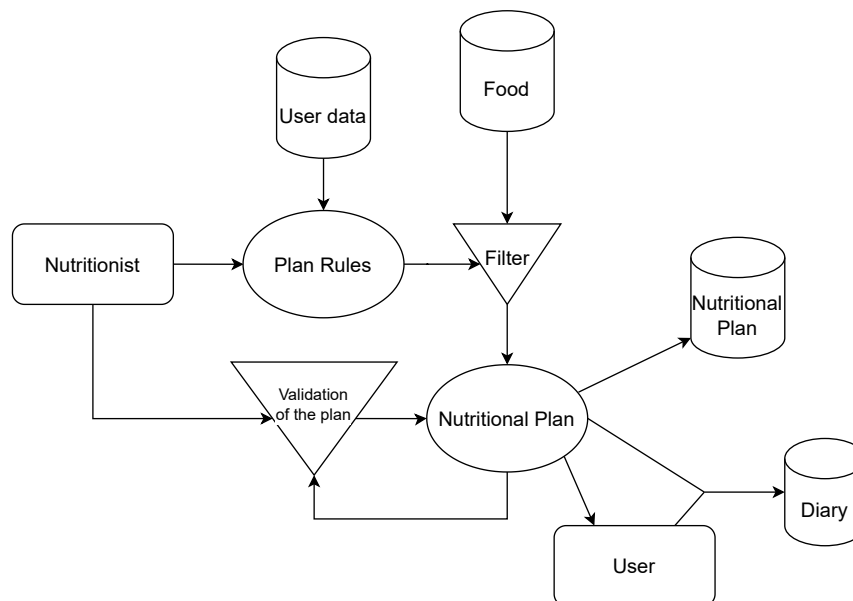


Figure 4.5: Flow Diagram of the Nutrition Plan Generation.

4.6 Recommendation System

The recommendation system's purpose is to fill out the analyzed patient's nutritional plan in a way that generates meal varieties without breaking the rules imposed by the responsible nutritionist. Making this task less repetitive for the nutritionist and more enjoyable for the patient.

At the beginning of the project it was imagined that this recommendation system should use some kind of artificial intelligence technique to infer a nutritional plan adapted to the patient profile. During the development of the system it was decided to use a rule-based approach that will allow to generate suggestions and send alerts to the nutritionist taking into account the limits and preferences established.

The steps of the RS below will all be performed internally by the system from the moment the nutritionist asks the system to fill in the rest of the nutritional plan, without interference at any point in the process. The steps are:

1. Apply filter to remove dishes containing food that the patient cannot consume;
2. Apply filter to remove dishes that have foods that the current nutritional plan prohibits from consuming;
3. Generate a patient consumption history list;
4. Analyze the foods on the consumption list and specified preferences, if any, to generate a table of possible patient preferences;
5. Apply the filter to remove foods that cannot be consumed in the preference table resulting in a preference table with only available foods;
6. Collect how many meals the nutritional plan has per day and the types defined for each meal;
7. Filter the dishes that are suitable for each type of meal;

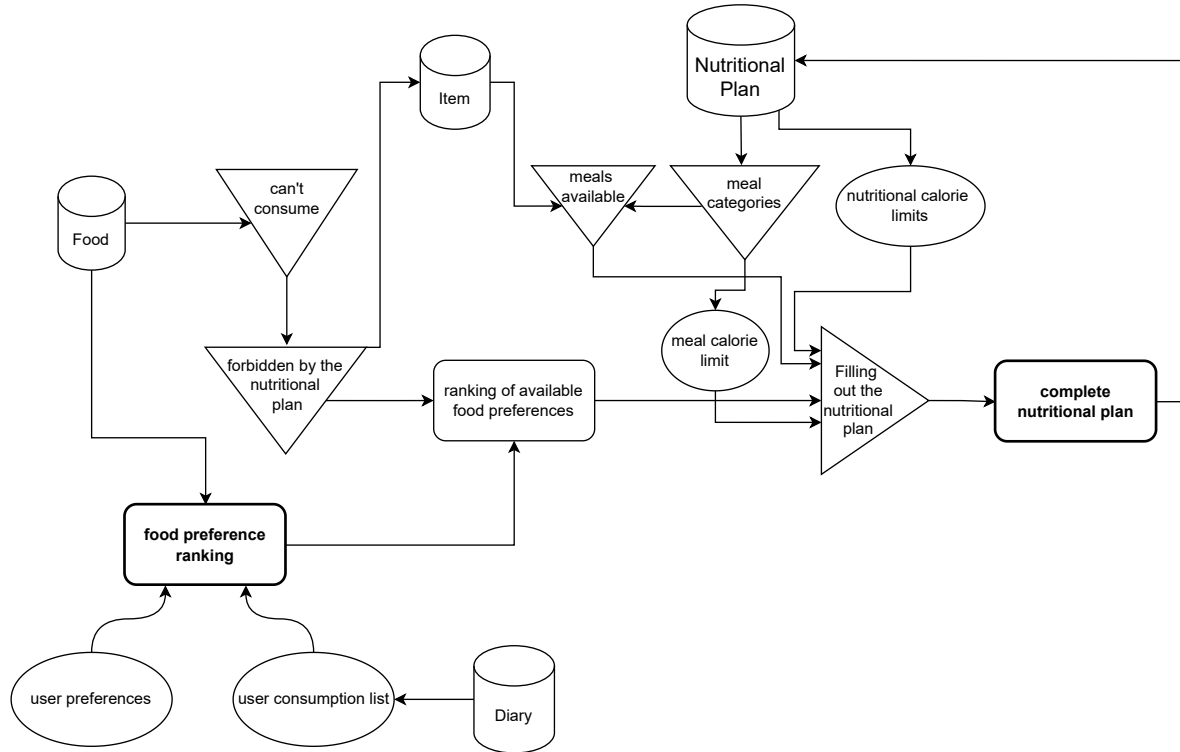


Figure 4.6: Recommendation System process overview.

8. Fill in the remaining meals of the nutritional plan, based on the table of preferences, in a varied way and meeting the imposed nutritional limits, which are the limits of calories, lipids, proteins, and the others mentioned earlier in this article.

In order to make the visualization of the process of the RS clearer and more objective, you can see the diagram in the Figure 4.6, which brings together all the steps mentioned above.

It is important to note that the steps 1 and 2 are simple filters in querying the meals in the database.

Step 3 will fetch only the foods consumed in the database table Diary.

The step 4 is to analyze the patient's food preferences and also the frequency that these appear in the diary to try to make a prediction of the meals that the patient is most likely to consume.

Step 5 will generate a food preference table with only the foods available for consumption, which means removing all forbidden foods from the preference table resulting from step 4.

The step 6 requires that the nutritionist has completed at least one day of the patient's nutritional plan, so it's a determining requirement for the RS to be able to be executed.

Step 7 is a filter on the meals available after steps 1 and 2 for each result from step 6.

And finally the step 8 is just filling the database taking into account the results of the steps 7 and 5, plus the calorie limits of the nutritional plan and the meal type, which are obtained in step 6.

4.7 Conclusion

This chapter provided a detailed explanation of the software modeling for the AquaVitae application. The User Stories and Use Cases, which play an essential role in the Requirements Engineering phase, were thoroughly presented and demonstrated the wide range of functionalities of the system. These encompass managing food information, designing personalized nutritional plans, and many other facets. Despite having identified all necessary features that the system must cover to be completed in this first stage of the project, it wasn't possible to fully implement all of them. This was due to technical issues that the frontend team faced, which caused a delay in the overall project.

The Entity Relationship Diagram (ERD) gave a deep understanding of the intricate interconnections between various tables and data points, highlighting the complexity and meticulous nature of the system.

In the Architecture section, the application of the SOA model to the project was emphasized, showcasing its advantages in developing reusable and interoperable software components.

The Main Flow section outlined a high-level operation of the system, illustrating how the nutritionist, the nutritional rules, and the user interact within the nutritional plan generation process.

Finally, the Recommendation System (RS) was detailed. This component is critical in enhancing the patient's experience by offering varied and personalized food recommendations.

All these elements combined contribute to the development of a robust, effective, and user-friendly system. The goal is to deliver maximum satisfaction to nutritionists and patients while prioritizing the personalization and adaptability of nutritional plans.

In conclusion, this chapter underscored the significance of software modeling in the successful development of the AquaVitae application. It also shed light on the importance of each component in improving user engagement and satisfaction, which is crucial for the app's adoption and success in real-world settings. The unforeseen challenges that arose during this stage of the project have provided valuable insights for future development, reaffirming the importance of adaptive and resilient project management.

Chapter 5

Backend Development

In this chapter the implementation of the backend is described, highlighting the most relevant points of the implementation. In order to have a clear understanding of the project development process, we will divide the chapter into three main sections that are *Overview of Technologies Used* (5.1), *Nutritional Management System Development* (5.2) and finally *Development of the Recommendation System* (5.3). The front-end development of the nutritional management system was developed at the same time by other master student. This allowed to do team work and develop some skills in collaborative programming.

Although there is a concern to make as clear as possible all the points that will be covered here, it is important to point out that reading this chapter requires a more advanced level of knowledge about programming.

5.1 Overview of Technologies Used

To understand some aspects of the following sections it's important to highlight which technologies were used and the reason for these choices, because they directly impact the way the software and the recommendation system is developed.

5.1.1 Programming Language

As a programming language it was decided to use *Python* [35] because of the previous knowledge that the person responsible for the backend development has about the language, and also because it is a great language for developing artificial intelligence and RS, which is also one of the goals of this project.

To facilitate the development environment either individually or with an eventual increase of developers working on the project the *Pipenv* [36] was chosen for versioning control of libraries for the construction of the application, because as they themselves describe the tool "*Pipenv is primarily meant to provide users and developers of applications with an easy method to setup a working environment*".

5.1.2 Database

In order to support this project, we decided to implement a relational database. The nature and volume of the data involved allows for establishing relationships between various data points, a feature that proved beneficial during the system's development. For our database management needs, we chose PostgreSQL. Not only is it an open-source tool and widely adopted within the industry, with notable users like Uber, Netflix, and Instagram (according to [37], a renowned site that tracks the technologies companies employ), it also allows the project team to explore and master this technology, thus expanding our professional competencies. This choice ensures reliable data management while bolstering the skills of the developer team.

With the decision to use PostgreSQL [38] as the project's database, it was necessary to look for some ORM to facilitate the development. For this was chosen to use SQLAlchemy [39] for also being open source and be the first choice of many developers when it comes to ORM in Python, and a great document that corroborated this choice was the paper referred in [40].

Together with SQLAlchemy [39] for migration control and database versioning it was

decided to use Alembic [41] taking into consideration the following self description: "*Alembic is a lightweight database migration tool for usage with the SQLAlchemy Database Toolkit for Python*".

5.1.3 Web Framework

The web framework for the construction of the backend that was suggested by the original proposal of the project, which can be seen in the Appendix ??, was the *FastAPI* [42] and it was decided to use it because it is a recent technology with great potential, according to some online articles such as the ones presented in [43] and [44], and also by a survey in [45]. Which, according to these articles, performs better than the already established *Flask* and *Django* in the Python development world.

With this defined, we also used all the other tools/libraries suggested by the framework such as *Pydantic* [46] and others, because this is how the technology was designed and projected to be used.

5.2 Nutritional Management System Development

In chapter 3 it was possible to observe some benefits of the REST protocol and SOA for the development of web systems through articles [28], [29] and [30]. These references support the decision to use them in the development of the application.

Therefore this section will discuss and demonstrate points that are important for understanding how the system was developed, so some parts can be left out of this report, but it is possible and recommended to see the complete code of the Aquavitae project in the GitHub repository [47] in the *aquavitae-thesis* branch that represents the project described in this dissertation and will remain unchanged.

Before starting to explain the code developed, it is important to point out that although most of the code produced in this project was authored by the developer, in some parts there was inspiration from code available in the documentation and GitHub of the

technologies used, which have already been mentioned above, such as *FastAPI* [42], *Pydantic* [46], *SQLAlchemy* [39] and so on, and also forums, being *Stack Overflow* [48] the main one used in this project.

Note also that the naming pattern used for directories, files and even in the code, was thought to make the code easier to read and understand for the developer and was based on the book [49].

5.2.1 Organization

A good project organization is fundamental for a fast application development because it enables the developer to have the notion of where each part of the system can be located and here the SOA has an important role and a big contribution.

In the Figure 5.1 you can see the following directories and files:

- Directory *alembic*: This is a standard database versioning directory that already explained in the 5.1.2 sub-section and contains all necessary configuration files as well as all project database migrations;
- Directory *src*: This is the directory that contains all the development files of the system and will be the focus of detail in this section;
- Directory *test*: In this directory you will find some parts of the frameworks for the development of the automated tests of the system, but this will be explained in more detail in chapter 6;
- File *config.py*: This file will consume the settings from the *.conf* file (which are in plain text) and will format and convert them to the correct format that should be readable for the entire system. It is possible to see the content of the file in the Figure 5.2.

Inside the *src* directory, as you can see in the 5.3 image, you can find the following items:

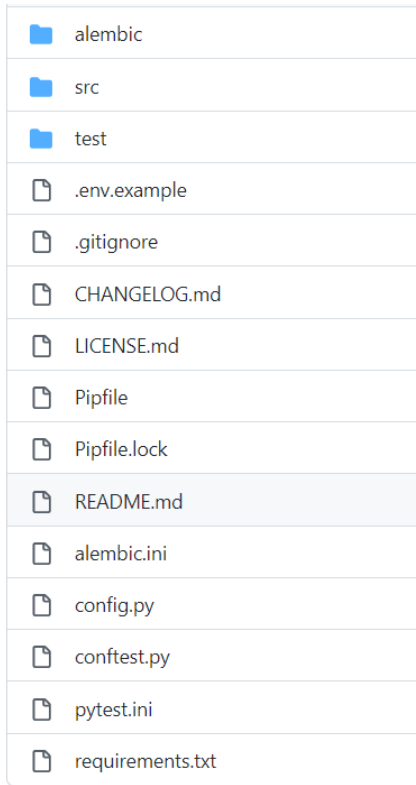


Figure 5.1: Project Organization
aquavitae-app

```

1  import os
2
3  from dotenv import load_dotenv
4
5  load_dotenv()
6
7  # ----- DEFAULT APP CONFIGURATION ----- #
8  APP_ENV = os.getenv("APP_ENV")
9  APP_TZ = os.getenv("APP_TZ")
10 APP_PORT = int(os.getenv("APP_PORT"))
11
12 CORS_ORIGINS = os.getenv("CORS_ORIGINS").split(",")
13
14 ROOT_DIR = os.path.dirname(os.path.abspath(__file__))
15
16 # ----- DEFAULT TEST CONFIGURATIONS ----- #
17 TEST_DATABASE_NAME = os.getenv("TEST_DATABASE_NAME")
18
19 # ----- DEFAULT DATABASE CONNECTION ----- #
20 DATABASE_CONNECTION = os.getenv("DATABASE_CONNECTION")
21 DATABASE_USERNAME = os.getenv("DATABASE_USERNAME")
22 DATABASE_PASSWORD = os.getenv("DATABASE_PASSWORD")
23 DATABASE_HOST = os.getenv("DATABASE_HOST")
24 DATABASE_PORT = int(os.getenv("DATABASE_PORT"))
25 DATABASE_NAME = os.getenv("DATABASE_NAME") if APP_ENV != "test" else TEST_DATABASE_NAME
26
27 # ----- DEFAULT SECRET KEY ----- #
28 TOKEN_SECRET_KEY = os.getenv("TOKEN_SECRET_KEY")
29 TOKEN_EXPIRATION_MINUTES = int(os.getenv("TOKEN_EXPIRATION_MINUTES"))
30 TOKEN_ALGORITHM = os.getenv("TOKEN_ALGORITHM")

```

Figure 5.2: File *config.py*

- Directory *core*: In this directory is found a series of other directories that contain code fundamental to the operation of the entire system, such as *constants*, *middlewares*, *types* and others, some of which will be discussed in more detail below.
- Directory *modules*: In the modules directory you will find all the modules/services of the system, divided into:
 - Module *app*: Here is the central module of the application, where all the endpoints that must be available and all the entities that must exist are grouped together;
 - Module *domain*: This directory contains all the modules that are part of the system but that are not necessary for the system to exist. In other words, the system can be run without some of these modules being present, however, for

all of the system's functionality to be present, the modules belonging to this directory must be fully functional;

- Module *infrastructure*: In this directory you will find the modules that are necessary for the system to exist, which so far are *auth* for authentication, *database* where you will find everything related to the database, which will have a special section for it below, and finally the *user* module.
- Directory *static*: So far its function is only to store images, such as the user's profile picture and body photos, which is part of a business rule in the system.
- File *main.py*: This file is where we run the entire application, however due to the layered organization that was used in this project, which will become clearer as we progress in this chapter, it allows it to be a small file as can be seen in the Figure 5.4. And we can observe:
 - Line 14 defines the application;
 - Line 17 adds middleware for *CORS*;
 - Line 25 adds middleware to limit the allowed size of a request;
 - Line 28 and 38 to set standard error messages for the system, which will be more detailed in the System Errors section below;
 - Line 35 makes all routes from the application available to the system;
 - Line 41 onwards runs the application through the *uvicorn*.

And finally inside a module the organization, as can be seen in the Figure 5.5, is as follows:

- Directory *controllers*: The directory concentrates all the endpoints that are respective to the module;
- Directory *dto*: Data Transfer Object (DTO) is responsible for hosting all the data that is expected to be received or sent to the client according to the standards defined by *pydantic* and *FastAPI*;

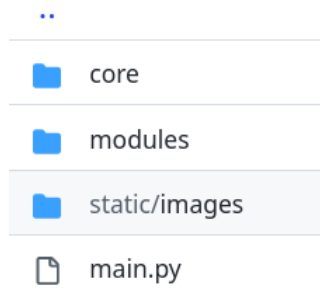


Figure 5.3: Directory *aquavita-app/src*

- Directory *entities*: All module entities must be defined here using the ORM used by the project;
- Directory *repositories*: All queries to the database must pass through this layer as an intermediary to access the database;
- Directory *services*: All endpoint business rules are developed at this layer;
- Directory *tests*: This directory contains all the tests of the module in question;
- File `__init__.py`: This file is where the module is defined, and where the endpoints that will be available and the entities that are to be created are defined and grouped.

5.2.2 Module Development

In this stage the development process of a single module will be described, from the beginning to the end, since all the others will follow the same logic. For this example we will use the *food* module.

The first step is to create the entity that will be developed, so the first layer to be developed is *entities*. You can see the contents of this directory in the *food* module through the Figure 5.6, notice that there are two files `*_entity.py`. This means that for this module to exist these tables must be present in the system.

Now looking at the content inside the file *food_entity.py* through the images 5.7 and 5.8, it is possible to observe the following points:

```

1  import os
2  import time
3
4  import uvicorn
5  from fastapi import FastAPI
6  from starlette.middleware.cors import CORSMiddleware
7
8  from config import APP_ENV, APP_PORT, APP_TZ, CORS_ORIGINS
9  from src.core.common.custom_error_response import custom_error_response
10 from src.core.handlers.http_exceptions_handler import HttpExceptionsHandler
11 from src.core.middlewares.limit_upload_size import LimitUploadSize
12 from src.modules.app import app_routers
13
14 app = FastAPI(title="Aquavitae App", version="0.0.1")
15
16 # CORS
17 app.add_middleware(
18     CORSMiddleware,
19     allow_origins=CORS_ORIGINS,
20     allow_credentials=True,
21     allow_methods=["*"],
22     allow_headers=["*"],
23 )
24
25 app.add_middleware(LimitUploadSize, max_upload_size=50_000_000) # ~50MB
26
27 # Register all custom exception handler
28 HttpExceptionsHandler(app)
29
30 # Set the default timezone of the application
31 os.environ["TZ"] = APP_TZ
32 time.tzset()
33
34 # Register all routers
35 app.include_router(app_routers)
36
37 # Modifies the error response pattern
38 custom_error_response(app)
39
40
41 if __name__ == "__main__":
42     uvicorn.run(
43         "src.main:app",
44         host="0.0.0.0",
45         port=APP_PORT,
46         reload=bool(APP_ENV != "production"),
47         use_colors=True,
48     )

```

Figure 5.4: File *src/main.py*

- *Line 11:* The table is declared, which inherits the *BaseEntity* of which will be detailed further in the sub-section 5.2.5;
- *Line 12-19:* The specific fields of the food table are declared. Note that they are typed, although Python doesn't require it, we have adopted this as standard and it will be present throughout the work, as it speeds up development and serves as

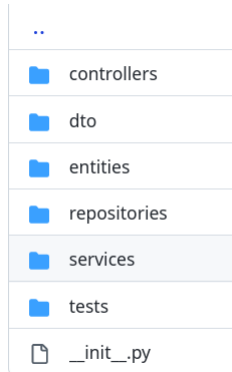


Figure 5.5: Directory *src/modules/domain/food*

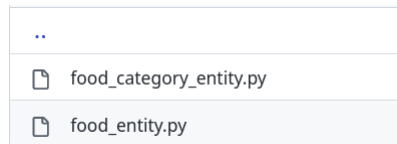


Figure 5.6: Directory *src/modules/domain/food/entities*

documentation for the project as well;

- *Line 21-24*: An example of how to declare the one-to-many relationship that the table has;
- *Line 26-37*: An example of how to declare which tables relate to the food table, so that ORM can do the mapping. In this example it is all many to one;
- *Line 39-62*: Declaring how the class is created.

With the entity defined, the next step is to create the repository layer which we can see from the Figure 5.9, and it is important to note the following points:

- The repository layer is for all queries from the module to the database to be defined here, however simple queries are well served by the *BaseRepository* created for this project and which will be detailed in the sub-section 5.2.7;
- *Line 05*: The Food module's repository class that inherits the *BaseRepository* is defined;

```

1  from dataclasses import dataclass
2
3  from sqlalchemy import Column, Float, ForeignKey, String
4  from sqlalchemy.dialects.postgresql import UUID
5  from sqlalchemy.orm import relationship
6
7  from src.modules.infrastructure.database.base_entity import BaseEntity
8
9
10 @dataclass
11 class Food(BaseEntity):
12     description: String = Column(String(255), nullable=False)
13     proteins: Float(2) = Column(Float(2), nullable=False)
14     lipids: Float(2) = Column(Float(2), nullable=False)
15     carbohydrates: Float(2) = Column(Float(2), nullable=False)
16     energy_value: Float(2) = Column(Float(2), nullable=False)
17     potassium: Float(2) = Column(Float(2), nullable=False)
18     phosphorus: Float(2) = Column(Float(2), nullable=False)
19     sodium: Float(2) = Column(Float(2), nullable=False)
20
21     food_category_id: UUID = Column(
22         UUID(as_uuid=True), ForeignKey("food_category.id", ondelete="CASCADE"), nullable=False
23     )
24     food_category = relationship("FoodCategory", back_populates="foods")
25
26     specificities = relationship(
27         "Specificity", back_populates="food", uselist=True, cascade="all, delete-orphan"
28     )
29     items = relationship(
30         "ItemHasFood", back_populates="food", uselist=True, cascade="all, delete-orphan"
31     )
32     forbidden_in_nutritional_plans = relationship(
33         "ForbiddenFoods", back_populates="food", uselist=True, cascade="all, delete-orphan"
34     )

```

Figure 5.7: **Part 01:** File `src/modules/domain/food/entities/food_entity.py`

- *Line 07:* The class that was inherited is started.

At this point in the module development the *services*, *controllers* and the DTO layers should be worked on together, however for explanatory purposes this order will be followed.

In the *services* there will be a lot of variation because it will depend on the business rule of each module, however some points remain the same, which we will use the Figure 5.10 to demonstrate:

- *Line 20:* The repository of the module to be used in the service layer must be started;
- *Line 23 and 30:* Define the types of data it expects to receive from the *controller* layer. In this case the DTO it wants to receive;


```

35     can_eat_at = relationship(
36         "FoodCanEatAt", back_populates="food", uselist=True, cascade="all, delete-orphan"
37     )
38
39     def __init__(
40         self,
41         description: String(255),
42         proteins: Float(2),
43         lipids: Float(2),
44         carbohydrates: Float(2),
45         energy_value: Float(2),
46         potassium: Float(2),
47         phosphorus: Float(2),
48         sodium: Float(2),
49         food_category_id: UUID,
50         *args,
51         **kwargs
52     ):
53         super().__init__(*args, **kwargs)
54         self.description = description
55         self.proteins = proteins
56         self.lipids = lipids
57         self.carbohydrates = carbohydrates
58         self.energy_value = energy_value
59         self.potassium = potassium
60         self.phosphorus = phosphorus
61         self.sodium = sodium
62         self.food_category_id = food_category_id

```

Figure 5.8: **Part 02:** File *src/modules/domain/food/entities/food_entity.py*

```

1  from src.modules.infrastructure.database.base_repository import BaseRepository
2  from src.modules.domain.food.entities.food_entity import Food
3
4
5  class FoodRepository(BaseRepository[Food]):
6      def __init__(self):
7          super().__init__(Food)

```

Figure 5.9: File *src/modules/domain/food/repositories/food_repository.py*

- *Line 23 and 31:* Define the type of data it expects to return to the *controller* if there is no error. In this case the DTO will be returned.
- *Line 27 and 34-39:* Return the correct data to the *controller*.

In the *controller* layer there is not much variation as the previous layer, being only necessary to declare the endpoint following some patterns and send to the *service* the data as expected. For example, we will use the Figure 5.11 and highlight the following points:

- By definition of *FastAPI* the *controller* layer cannot be inside a class, as the class

```

18 class FoodService:
19     def __init__(self):
20         self.food_repository = FoodRepository()
21
22     # ----- PUBLIC METHODS -----
23     async def create_food(self, food_dto: CreateFoodDto, db: Session) -> Optional[FoodDto]:
24         new_food = await self.food_repository.create(food_dto, db)
25
26         new_food = await self.food_repository.save(new_food, db)
27         return FoodDto(**new_food.__dict__)
28
29     async def get_all_food_paginated(
30         self, pagination: FindManyOptions, db: Session
31     ) -> Optional[PaginationResponseDto[FoodDto]]:
32         [all_food, total] = await self.food_repository.find_and_count(pagination, db)
33
34         return create_pagination_response_dto(
35             [FoodDto(**food.__dict__) for food in all_food],
36             total,
37             pagination["skip"],
38             pagination["take"],
39         )

```

Figure 5.10: File `src/modules/domain/food/services/food_service.py`

wouldn't be initialized;

- *Line 25:* The route to the food endpoints is defined;
 - It was defined that the default for prefixing the endpoints of a table would be the same as the table name in lowercase;
 - If the table name were compound, there would be hyphen ("-") separation.
- *Line 31 and 43:* Declaration of the endpoint path. No pattern has been set for the path name, however it should be something descriptive so that the code is able to self document;
- *Line 33 and 44:* Define what the endpoint will return if the request succeeds. With this type set correctly the endpoint documentation made by the framework is done correctly, more details will be given in the sub-section 5.2.6;
- *Line 34 and 46:* Define the authentication required to access the endpoint;
 - If there isn't any *Auth*, the endpoint is open for everyone;
 - If there is only *Auth* this endpoint is accessible to everyone connected to the system;

- If there is one or more user roles in *Auth* only users who have that role can access the endpoint;
- It was decided not to use hierarchical user roles, meaning that the endpoints available for lower positions would be available for higher positions, as this wouldn't allow this flexibility.
- *Line 37 and 49-52*: Defining the data needed for each endpoint. Here it is extremely important to type the data, because together with *Pydantic* it is possible to block requests that have missing or unexpected data;
- *Line 45*: Remove data from the response model that wasn't filled in, in order to reduce the size of the response.

```

25 food_router = APIRouter(tags=["Food"], prefix="/food")
26
27 food_service = FoodService()
28
29
30 @food_router.post(
31     "/create",
32     status_code=HTTP_201_CREATED,
33     response_model=FoodDto,
34     dependencies=[Depends(Auth([UserRole.ADMIN, UserRole.NUTRITIONIST]))],
35 )
36 async def create_food(
37     request: CreateFoodDto, database: Session = Depends(get_db)
38 ) -> Optional[FoodDto]:
39     return await food_service.create_food(request, database)
40
41
42 @food_router.get(
43     "/get",
44     response_model=PaginationResponseDto[FoodDto],
45     response_model_exclude_unset=True,
46     dependencies=[Depends(Auth([UserRole.ADMIN, UserRole.NUTRITIONIST]))],
47 )
48 async def get_all_food_paginated(
49     pagination: FindManyOptions = Depends(
50         GetPagination(Food, FoodDto, FindAllFoodQueryDto, OrderByFoodQueryDto)
51     ),
52     database: Session = Depends(get_db),
53 ) -> Optional[PaginationResponseDto[FoodDto]]:
54     return await food_service.get_all_food_paginated(pagination, database)

```

Figure 5.11: File *src/modules/domain/food/controllers/food_controller.py*

Last but not least are the DTOs. As each module can have numerous DTO it is divided into files where each file has DTO for specific functions in the system as you can see in the Figure 5.12.



Figure 5.12: Directory *src/modules/domain/food/dto/food*

For example, the files *food_dto.py* and *update_food_dto.py* will be used to demonstrate how to define them correctly in order to be useful for both endpoint typing and documentation, and also as an extra layer of security for the system.

The file *food_dto.py* can be viewed through the Figure 5.13 and the following points can be highlighted:

- *Line 10*: The class is declared by inheriting *BaseDto* which is a DTO with default data that all entities that inherit attributes from *BaseEntity* have;
- *Line 11-19*: Declaring the attributes of *glsDTO* according to the *Pydantic* standards;
- *Line 22-23*: The preference for filling the *food_category* field is the *FoodCategoryDTO* object, however if not present this field is filled by the *UUID* present in *food_category_id*.

And in the file *update_food_dto.py* in the Figure 5.14, you notice a few other different points, which are:

- *Line 7*: The class now inherits the *BaseModel* from *Pydantic*;
- *Line 8-16*: The data that will be accepted by this DTO is just that. Their presence in the request is all optional as you can see, but if they are present they must only have the data type that has been declared, anything else will be blocked;
- *Line 19*: Attributes that are not declared in this class will be blocked if this DTO is being used.

```

1  from typing import Optional, Union
2  from uuid import UUID
3
4  from pydantic import condecimal, constr
5
6  from src.core.common.dto.base_dto import BaseDto
7  from src.modules.domain.food.dto.food_category.food_category_dto import FoodCategoryDto
8
9
10 class FoodDto(BaseDto):
11     description: Optional[constr(max_length=255)]
12     proteins: Optional[condecimal(decimal_places=2)]
13     lipids: Optional[condecimal(decimal_places=2)]
14     carbohydrates: Optional[condecimal(decimal_places=2)]
15     energy_value: Optional[condecimal(decimal_places=2)]
16     potassium: Optional[condecimal(decimal_places=2)]
17     phosphorus: Optional[condecimal(decimal_places=2)]
18     sodium: Optional[condecimal(decimal_places=2)]
19     food_category: Optional[Union[FoodCategoryDto, UUID]]
20
21     def __init__(self, **kwargs):
22         if "food_category" not in kwargs and "food_category_id" in kwargs:
23             kwargs["food_category"] = kwargs["food_category_id"]
24         super().__init__(**kwargs)
25
26     class Config:
27         orm_mode = True

```

Figure 5.13: File `src/modules/domain/food/dto/food/food_dto.py`

```

7  class UpdateFoodDto(BaseModel):
8      description: Optional[constr(max_length=255)]
9      proteins: Optional[condecimal(decimal_places=2)]
10     lipids: Optional[condecimal(decimal_places=2)]
11     carbohydrates: Optional[condecimal(decimal_places=2)]
12     energy_value: Optional[condecimal(decimal_places=2)]
13     potassium: Optional[condecimal(decimal_places=2)]
14     phosphorus: Optional[condecimal(decimal_places=2)]
15     sodium: Optional[condecimal(decimal_places=2)]
16     food_category_id: Optional[UUID] = Field(alias="food_category")
17
18     class Config:
19         extra = Extra.forbid

```

Figure 5.14: File `src/modules/domain/food/dto/food/update_food_food_dto.py`

5.2.3 Error Handling

During the development of the system it was noticed that the only way provided by the framework to generate error messages would be manually through the *HTTPException* as can be seen in the Figure 5.15 extracted from the documentation of the *FastAPI*.



```

from fastapi import FastAPI, HTTPException

app = FastAPI()

items = {"foo": "The Foo Wrestlers"}

@app.get("/items/{item_id}")
async def read_item(item_id: str):
    if item_id not in items:
        raise HTTPException(status_code=404, detail="Item not found")
    return {"item": items[item_id]}

```

Figure 5.15: Example of handling errors extracted from the *FastAPI* documentation.

This wasn't desirable for the system since it would require the developer to memorize the HTTP error codes, besides not defining a standard for the data that would be returned in the errors generated by the system. Therefore a standardization was developed that involved some steps, which are:

1. Define which data should be present in all error messages. The data that was considered important is in the Figure 5.16;
2. Define the different error types that the system will use, observable by the Figure 5.17;
 - Note, in the image referenced above, that the only field required to generate these errors is a message (variable *msg*);
 - It is no longer necessary to memorize the HTTP error codes, only the name of each error;
 - The data that each error has is now standardized.
3. Declare a *Handler* so that when such an error is triggered, it is treated as an error and not just any message. Observable by the Figure 5.18.

By default the *FastAPI* error messages generated by the framework have the format present in the Figure 5.19.

```

7  class DetailResponseDto(BaseModel):
8      loc: List[str] = Field(title="Location")
9      msg: str = Field(title="Message")
10     type: str = Field(title="Error Type")
11
12
13  class ExceptionResponseDto(BaseModel):
14      detail: List[DetailResponseDto]
15      status_code: int = 422
16      timestamp: datetime = Field(title="Timestamp of the Request")
17      path: str = Field(title="Request Path")
18      method: str = Field(title="Request Method")
19
20     class Config:
21         extra = Extra.forbid

```

Figure 5.16: File *src/core/common/dto/exception_response_dto.py*

```

8  class BadRequestException(BaseExceptionType):
9      def __init__(self, msg: str, loc: List[str] = [], _type: str = None):
10         if _type is None:
11             _type = "bad_request"
12
13         self.status_code = HTTP_400_BAD_REQUEST
14         super().__init__(msg, loc, _type)
15
16
17  class UnauthorizedException(BaseExceptionType):
18      def __init__(self, msg: str, loc: List[str] = [], _type: str = None):
19         if _type is None:
20             _type = "unauthorized"
21
22         self.status_code = HTTP_401_UNAUTHORIZED
23         super().__init__(msg, loc, _type)
24
25
26  class ForbiddenException(BaseExceptionType):
27      def __init__(self, msg: str, loc: List[str] = [], _type: str = None):
28         if _type is None:
29             _type = "forbidden"
30
31         self.status_code = HTTP_403_FORBIDDEN
32         super().__init__(msg, loc, _type)
33
34
35  class NotFoundException(BaseExceptionType):
36      def __init__(self, msg: str, loc: List[str] = [], _type: str = None):
37         if _type is None:
38             _type = "not_found"
39
40         self.status_code = HTTP_404_NOT_FOUND
41         super().__init__(msg, loc, _type)

```

Figure 5.17: File *src/core/types/exception_types.py*

```

59         @self.app.exception_handler(BadRequestException)
60         @self.app.exception_handler(UnauthorizedException)
61         @self.app.exception_handler(ForbiddenException)
62         @self.app.exception_handler(NotFoundException)
63         async def custom_exceptions_handler(
64             request: Request, exc: BadRequestException
65         ) -> Response:
66             detail = deepcopy(exc)
67             delattr(detail, "status_code")
68
69             return Response(
70                 status_code=exc.status_code,
71                 content=json.dumps(
72                     self.global_exception_error_message(
73                         status_code=exc.status_code,
74                         detail=DetailResponseDto(**detail.__dict__),
75                         request=request,
76                     ).__dict__,
77                     default=JsonUtils.json_serial,
78                 ),
79             )
80
81         @staticmethod
82         def global_exception_error_message(
83             status_code: int,
84             detail: Union[DetailResponseDto, List[DetailResponseDto]],
85             request: Request,
86         ) -> ExceptionResponseDto:
87             if not isinstance(detail, List):
88                 detail = [detail]
89
90             return ExceptionResponseDto(
91                 detail=detail,
92                 status_code=status_code,
93                 timestamp=datetime.now().astimezone(),
94                 path=request.url.path,
95                 method=request.method,
96             )

```

Figure 5.18: **Part 1:** File `src/core/types/http_exception_handler.py`

Care was taken in the creation of the customizable errors to follow the pattern of these errors generated by the framework, however as can be seen in the Figure 5.16 the data *status_code*, *timestamp*, *path* and *method* were added to the body of the response.

Therefore it was desirable that the framework's error messages follow the same pattern as the customizable messages, as it would make it easier for the front-end to handle these responses if they were all the same. To make this possible two steps were necessary, which are:


```

{
  "detail": [
    {
      "loc": [
        "path",
        "item_id"
      ],
      "msg": "value is not a valid integer",
      "type": "type_error.integer"
    }
  ]
}

```

Figure 5.19: Example error message extracted from the FastAPI documentation.

1. It was necessary to define *Handlers* for the possible error types that the framework throws which are *StarletteHTTPException* and *RequestValidationError*, observable by the Figure 5.20;
2. For documentation purposes it was necessary to change the format of the body of the error responses on all the application endpoints. The code can be seen in the Figure 5.21.
 - With this standardization of error messages it isn't necessary to put in the documentation all the possible errors and their respective answers for an endpoint, since whatever the error is, the format is always the same, with only the content changing.
 - This code is invoked on line 38 of the file *main.py* from the Figure 5.4

5.2.4 Pagination

It was necessary to develop a pagination for the *GET* endpoints in order to avoid overloading the API when some table with a lot of data was consulted.

So that this effort wouldn't be repeated for all the endpoints of the system, a generic solution that could handle if not all, at least most of these endpoints was desirable.

```

28     def add_exceptions_handler(self):
29         @self.app.exception_handler(StarletteHTTPException)
30         async def http_exception_handler(request: Request, exc) -> Response:
31             return Response(
32                 status_code=exc.status_code,
33                 content=json.dumps(
34                     self.global_exception_error_message(
35                         status_code=exc.status_code,
36                         detail=DetailResponseDto(loc=[], msg=exc.detail, type="starlette_http_exception"),
37                         request=request,
38                     ).__dict__,
39                     default=JsonUtils.json_serial,
40                 ),
41             )
42
43         @self.app.exception_handler(RequestValidationError)
44         async def validation_exception_handler(
45             request: Request, exc: RequestValidationError
46         ) -> Response:
47             return Response(
48                 status_code=HTTP_422_UNPROCESSABLE_ENTITY,
49                 content=json.dumps(
50                     self.global_exception_error_message(
51                         status_code=HTTP_422_UNPROCESSABLE_ENTITY,
52                         detail=[DetailResponseDto(**detail) for detail in exc.errors()],
53                         request=request,
54                     ).__dict__,
55                     default=JsonUtils.json_serial,
56                 ),
57             )

```

Figure 5.20: **Part 2:** File `src/core/handlers/http_exceptions_handler.py`

It's possible to see how to add pagination to an endpoint in the *Line 50* of the Figure 5.11 already mentioned in previous sections. But we can take a closer look at this method in the Figure 5.22 and highlight the following points:

- *Line 17:* The *Entity* that the pagination is working with is used to generate the *order by*, where of the Query along with the columns and relationships that will be brought with it;
- *Line 18:* These are the columns and relationships that are allowed to be queried by this endpoint, any other column that tries to be queried will be blocked;
- *Line 19-20:* These are the columns that can be present in the *where* and *order by* of the Query respectively, and as mentioned in the previous item they also block any other columns that try to be queried or if they don't have the specified data format;
- From *Line 29* to *38* of the `__call__` method are all the variables that can be present in the endpoint's *Query*:

```

7  def custom_error_response(app: FastAPI):
8      if app.openapi_schema:
9          return app.openapi_schema
10     openapi_schema = get_openapi(
11         title=app.title,
12         version=app.version,
13         description=app.description,
14         routes=app.routes,
15     )
16
17     # not quite the ideal scenario but this is the best we can do to override the default
18     # error schema. See
19     from fastapi.openapi.constants import REF_PREFIX
20     import pydantic.schema
21
22     paths = openapi_schema["paths"]
23     for path in paths:
24         for method in paths[path]:
25             if paths[path][method]["responses"].get("422"):
26                 paths[path][method]["responses"]["422"] = {
27                     "description": "Validation Error",
28                     "content": {
29                         "application/json": {
30                             "schema": {"$ref": f"{REF_PREFIX}ExceptionResponseDto"}
31                         }
32                     },
33                 }
34
35     error_response_defs = pydantic.schema.schema(
36         (ExceptionResponseDto,), ref_prefix=REF_PREFIX, ref_template=f"{REF_PREFIX}{{model}}")
37
38     openapi_schemas = openapi_schema["components"]["schemas"]
39     openapi_schemas.update(error_response_defs["definitions"])
40     openapi_schemas.pop("ValidationError")
41     openapi_schemas.pop("HTTPValidationError")
42
43     app.openapi_schema = openapi_schema

```

Figure 5.21: File `src/core/common/custom_error_response.py`

- *Line 29*: The parameter *skip* is the number of pages that the request wants to skip;
- *Line 30*: The parameter *take* is the number of items that the request will return and is used in conjunction with the previous parameter;
- *Line 31*: The *search* parameter is where the "where" of the *Query* will be applied, and it is possible to specify the column ("*field*") and the value ("*value*");
- *Line 34*: The *sort* parameter is where the "order by" of the *Query* will be applied, and it is possible to specify the column ("*field*") and the order ("*by*");
- *Line 37*: With the parameter *columns* it is possible to specify which columns and relationships to bring in with the search;

- * Columns that are mandatory are not required to be specified;
- *Line 38*: With the parameter *search_all* the value (*value*) sent will be applied also to the "where" of the *Query*, similar to the parameter ("search"), however the search will be in all columns of the table.

```

14 class GetPagination(object):
15     def __init__(
16         self,
17         entity: E,
18         columns_query: C,
19         find_all_query: F = None,
20         order_by_query: O = None,
21     ):
22         self.entity = entity
23         self.columns_query = columns_query
24         self.find_all_query = find_all_query
25         self.order_by_query = order_by_query
26
27     def __call__(
28         self,
29         skip: int = Query(default=1, ge=1),
30         take: int = Query(default=10, ge=1, le=100),
31         search: Union[list[str], None] = Query(
32             default=None, regex=".*.*$", example=["field:value"]
33         ),
34         sort: Union[list[str], None] = Query(
35             default=None, regex=".*(ASC|DESC|\\+|\\-)$", example=["field:by"]
36         ),
37         columns: Union[list[str], None] = Query(default=None, regex=".*", example=["field"]),
38         search_all: Union[str, None] = Query(default=None),
39     ) -> FindManyOptions:
40         paging_params = PaginationUtils.generate_paging_parameters(
41             skip,
42             take,
43             search,
44             sort,
45             self.find_all_query,
46             self.order_by_query,
47         )
48
49         return PaginationUtils.get_paging_data(
50             self.entity,
51             paging_params,
52             search_all,
53             columns if columns else [],
54             self.columns_query,
55             self.find_all_query,
56         )

```

Figure 5.22: File *src/core/decorators/pagination_decorator.py*

The rest of the code is the formatting of the data to be sent to the endpoint of the

request and won't be detailed in this document.

But with this brief explanation it is already possible to observe and conclude that this *Pagination* not only covers most of the application endpoints, but also provides a security layer allowing only the data input that the developer wants.

The encapsulation of this solution also brings some benefits for the development of the project since, if a correction in the code is needed, all endpoints would receive the update, making the maintenance of the project easier and more pleasant.

5.2.5 Base Entity

In the Figure 4.2 in the database modeling section 4.3 of the modeling chapter 4 you can see that all the tables have four attributes in common, which are:

1. *id*: Primary key of the row;
2. *created_at*: Date of creation of the row;
3. *updated_at*: Last updated date of the row;
4. *deleted_at*: Date the row was "*deleted*".

* The column *deleted_at* exists because the software will support *soft delete*, that is, the data can remain in the database and only be marked as unusable. This strategy allows for data recovery if a deletion was done by mistake, and it helps maintain a historical record of operations. However, it can affect the scalability of the database over time, as data continues to accumulate even after deletion. Nonetheless, we opted to implement soft delete in this project due to its advantages in facilitating error correction and data auditing. It's a widely discussed feature with known pros and cons, but a detailed analysis of its suitability is outside the scope of this document.

Knowing that this data would be repeated throughout the system, as in the previous subsection, it was also desirable to have a generic solution for all the entities in the system,

in order to make the code cleaner and easier to maintain. Therefore, the *Base Entity*, observable in the Figure 5.23, was created and some functionalities implemented, which are:

- *Line 17*: The primary key is defined and on its creation if no value is defined a new one of type *uuid4* is generated;
- *Line 21*: The *deleted_at* column is marked as the column that defines the deleted via the *info* parameter;
- *Line 28-30*: To standardize the name of the tables in the database and avoid having to define the *__tablename__* in all entities, the functionality to generate the automatic name following the pattern of the whole name being in lower case and in case of compound names having a separation by *"_"*;
- *Line 33-38*: Define an event to be executed every time a new item proceeding from *BaseEntity* is inserted into the database;
- *Line 41-46*: Define an event to be executed every time an item proceeding from *BaseEntity* is updated in the database.

As this data could possibly be present in all responses to requests made to the system, a *BaseDto* was also made, observable by the Figure 5.24, to achieve the same goals as the *BaseEntity* already mentioned above.

5.2.6 Documentation

Backend documentation is important for developers because it makes it easy to see the endpoints, which data is accepted, which is required, and what possible answers the endpoint can return.

In this work there was care that all the endpoints of the system were well documented through all the DTO that was detailed in the previous subsections, and now it is possible to observe the final result through the images below.

```

13 @dataclass
14 class BaseEntity(Base):
15     __abstract__ = True
16
17     id: UUID = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
18     created_at: DateTime = Column(DateTime(timezone=True), nullable=False)
19     updated_at: DateTime = Column(DateTime(timezone=True), nullable=False)
20     deleted_at: DateTime = Column(
21         DateTime(timezone=True), nullable=True, info={"delete_column": True}
22     )
23
24     def __init__(self, *args, **kwargs):
25         super().__init__(*args, **kwargs)
26
27     # Generate __tablename__ automatically
28     @declared_attr
29     def __tablename__(self) -> str:
30         return "_".join(re.findall("[A-Z][^A-Z]*", self.__name__)).lower()
31
32
33 @event.listens_for(BaseEntity, "before_insert", propagate=True)
34 def set_before_insert(mapper, connection, target: BaseEntity) -> None:
35     if not target.created_at:
36         target.created_at = datetime.now()
37     if not target.updated_at or target.updated_at < target.created_at:
38         target.updated_at = target.created_at
39
40
41 @event.listens_for(BaseEntity, "before_update", propagate=True)
42 def set_before_update(mapper, connection, target: BaseEntity) -> None:
43     if target.deleted_at:
44         target.updated_at = target.deleted_at
45     else:
46         target.updated_at = datetime.now()

```

Figure 5.23: File *src/modules/infrastructure/database/base_entity.py*

```

12 class BaseDto(BaseModel):
13     id: UUID
14     created_at: Optional[datetime]
15     updated_at: Optional[datetime]
16     deleted_at: Optional[datetime]

```

Figure 5.24: File *src/core/common/dto/base_dto.py*

In the Figure 5.25, in item 1 it's possible to see how the request body should be composed to send to the endpoint */food/create*, with the data that is required and its respective types. In item 2 it's possible to see an example of what this body would look like in *JSON* format.

In the Figure 5.26, in item 1 it's possible to see the successful request response with all the attributes that must be in the response and the optional ones along with their respective types. In item 2 it's possible to see an example of what this body looks like in *JSON* format.

In the Figure 5.27, it's possible to see the possible variables available in the *Query* of the request to query the table with paging.

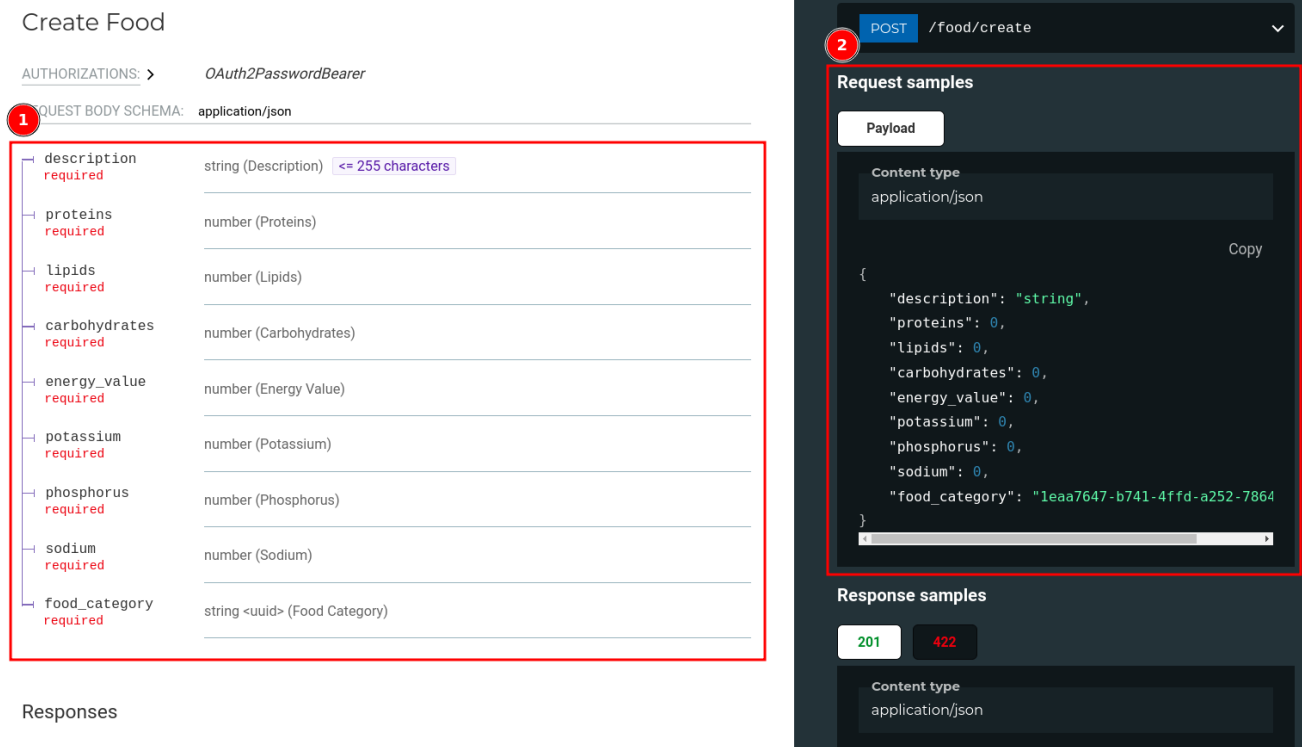


Figure 5.25: Create Food request body.

In the Figure 5.28 it's possible to see the successful request response to an endpoint that has pagination with all the attributes that must be in the response and the optional ones along with their respective types.

In the Figure 5.29 it's possible to see an example of what a successful response from an endpoint that has pagination in *JSON* format looks like.

And finally in the Figure 5.30 it is possible to observe the error response format for the whole system regardless of the endpoint or the error code and as in the previous images in item 1 the detailed response is observed and in item 2 in the format *JSON*.

The images above were taken only from two endpoints of the *Food* module, but it is enough to understand how the documentation was applied throughout the system and to conclude that this visual information of the endpoints facilitates the development of the front-end and even the backend.

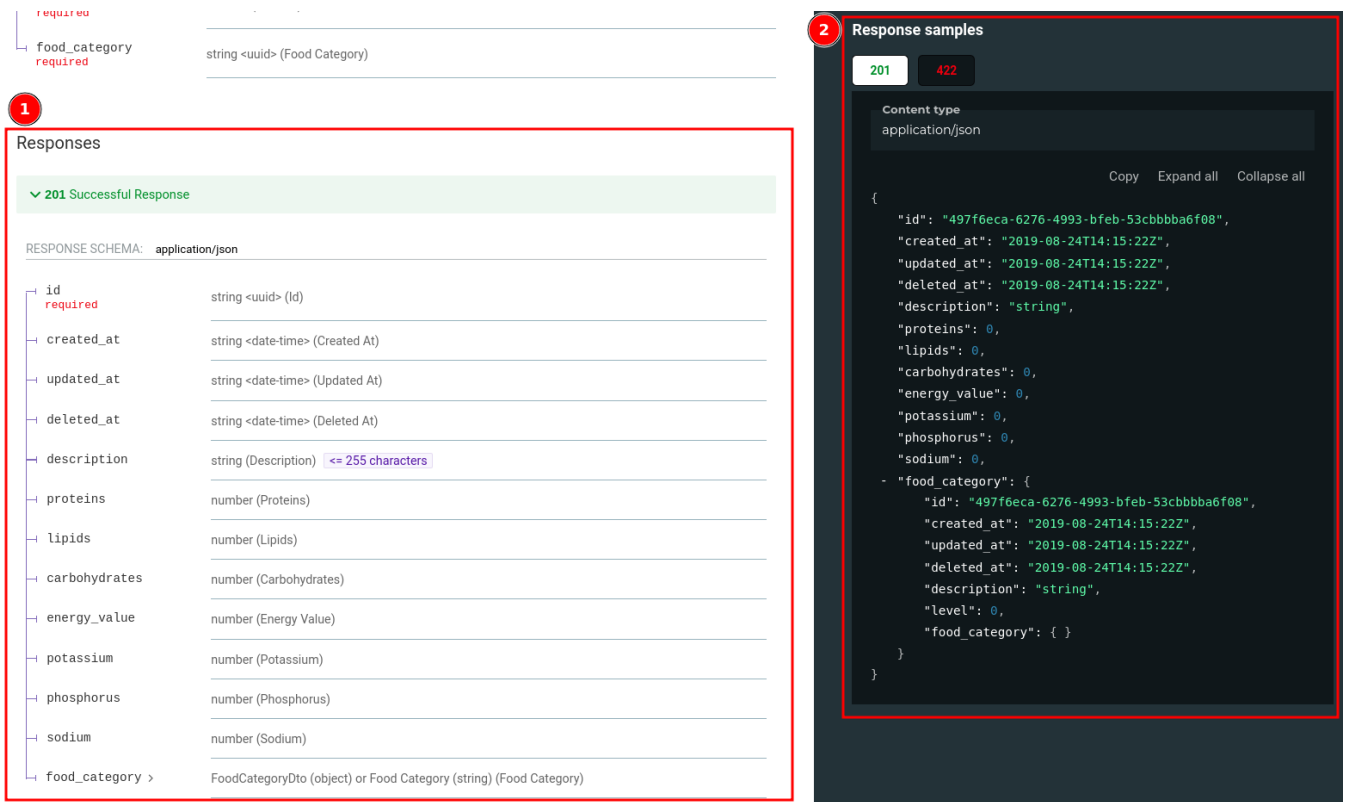


Figure 5.26: Create Food response body.

5.2.7 Generic Repository

Finally, the feature that took the most time in the development of the system was undoubtedly the construction of a generic repository in order to serve all tables in the system for the most common CRUD actions of a Web system, and therefore avoid repetition of code and effort throughout the system.

The details of the development of this feature won't be detailed here but it was expected that this generic repository would meet some requirements:

1. Insert new data in the table;
2. Save the changes to the database;
3. Search for a specific row;
4. Fetch multiple rows;

Get All Food Paginated

AUTHORIZATIONS: > OAuth2PasswordBearer

QUERY PARAMETERS

skip	integer (Skip) <code>>= 1</code> Default: <code>1</code>
take	integer (Take) <code>[1 .. 100]</code> Default: <code>10</code>
search	Array of strings (Search) <code>.*:.*\$</code> Example: <code>search=field:value</code>
sort	Array of strings (Sort) <code>.*(ASC DESC \+ \-)\$</code> Example: <code>sort=field:by</code>
columns	Array of strings (Columns) <code>.*</code> Example: <code>columns=field</code>
search_all	string (Search All)

Figure 5.27: Variables of an endpoint with pagination.

5. Update data;
6. Permanently delete data in the database;
7. Logically delete (soft delete) the data in the database;
8. Be able to construct *queries* with different parameters dynamically for all methods available in the generic repository.

The items 1 and 2 are relatively simple methods because it's just a matter of inserting data in the table that is being worked on and saving changes in the database respectively. The item 2 was made into a separate method because this way it's possible to perform and control transactions that must be made atomic to the database.

The item 3 required two methods to be completely satisfied, one method would try to find the data requested by the service layer and if found would return it, but if not found

Responses

✓ 200 Successful Response

RESPONSE SCHEMA: application/json

data ✓ required	Array of objects (Data)
Array [
id required	string <uuid> (Id)
created_at	string <date-time> (Created At)
updated_at	string <date-time> (Updated At)
deleted_at	string <date-time> (Deleted At)
description	string (Description) ≤ 255 characters
proteins	number (Proteins)
lipids	number (Lipids)
carbohydrates	number (Carbohydrates)
energy_value	number (Energy Value)
potassium	number (Potassium)
phosphorus	number (Phosphorus)
sodium	number (Sodium)
food_category >	FoodCategoryDto (object) or Food Category (string) (Food Category)
]	
count required	integer (Count)
limit required	integer (Limit)
current_page required	integer (Current Page)
next_page	integer (Next Page)
prev_page	integer (Prev Page)
last_page required	integer (Last Page)

Figure 5.28: Get Food detailed response body with pagination.

it wouldn't return anything and the service layer would decide what to do. The second method is similar to the first but if it didn't find the requested data it would issue an error and stop execution. This is useful because, in some cases, this would be the default behavior and, to avoid repeating code issuing errors that might not be standardized, this was relayed in the repository layer.

In the item 4 two methods were also necessary, both of them search the database with the query that was assigned and may bring an empty list or with several rows of the table

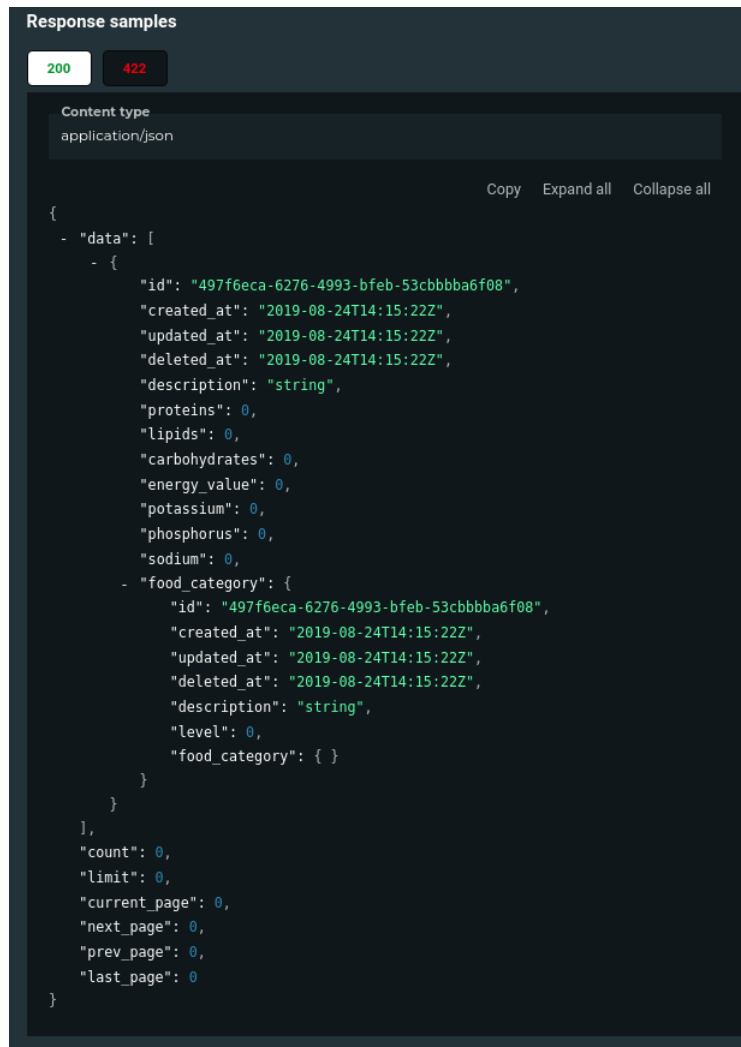


Figure 5.29: Get Food response body with pagination in *JSON* format.

worked, however, one method also performs the count how many elements the query has without the interference of *offset* and *limit*, coming from the parameters *skip* and *take* from *pagination* respectively, if present in the query.

On 5 and 6 there is not much to say, they are simple methods that perform permanent updating and deletion of data in the database respectively.

The item 7 required the most effort because in order *soft delete* to support it was necessary to overcome some challenges:

1. Cascading deletion will no longer work;

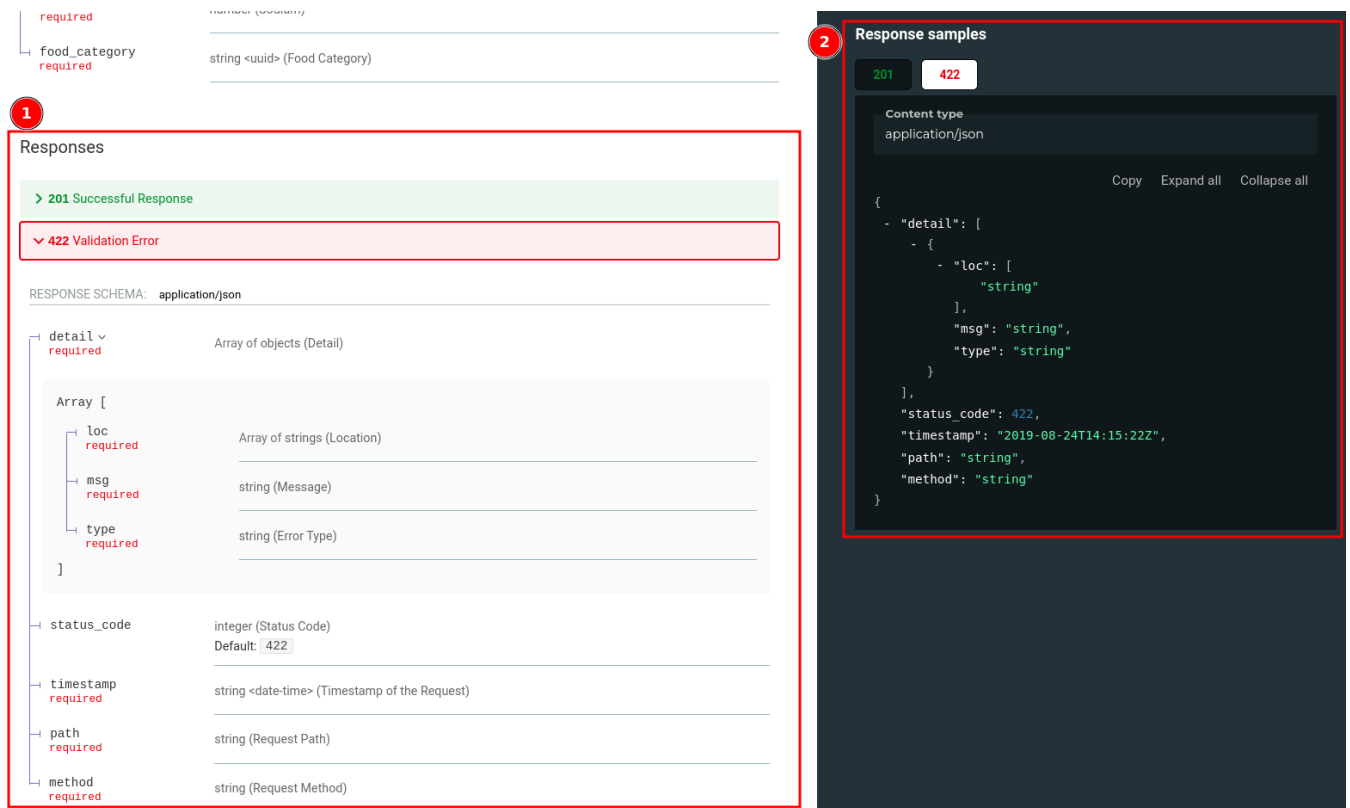


Figure 5.30: Error response body in *JSON* format.

- A method was developed to apply *soft delete* to all relationships that depended on the entity that *soft delete* was being applied to.
2. Indexes wouldn't work anymore since there might be data in the table that was deleted and a new one with the same value couldn't be inserted;
 - As the *PostgreSQL* database was used, to solve this problem it was necessary to create compound indexes, that is, the value to be *unique* would also depend on the *deleted_at* column, one of the factors that corroborated for the column to be of type *Date time*, because if it were *Boolean* the index wouldn't work correctly either.
 3. All queries should contain a filter so as not to bring deleted rows;
 - It wasn't desirable that the programmer, when constructing the parameters for

the formation of the *Query*, included the filter manually, because this would be exhausting and easily forgotten and would lead to undesirable data leakage that would be easily noticed by the user or someone testing the system;

- To solve this problem, as you can see in the Figure 5.31, an event was created to insert, in all queries launched by the system, a filter to remove deleted lines from the query if the column that represented deleted lines wasn't already included in the *where clauses*, otherwise the developer would have already dealt with this column and the filter wouldn't interfere;
- The filter for not bringing deleted rows couldn't be applied to the entity relations that would be searched, which forced a manual check and adjustment in code, generating *overhead* in the consult queries;
- To create or update an entity it was necessary to add a check in the relationships in order to verify whether the data was valid or represented a deleted row.

```
31 # add filter to remove deleted entities by default every time a query of this class is executed
32 @event.listens_for(Query, "before_compile", retval=True)
33 @pause_listener
34 def no_deleted(query: Query) -> Query:
35     columns = (
36         []
37         if not isinstance(query.column_descriptions[0]["entity"], DeclarativeMeta)
38         else get_columns(query.column_descriptions[0]["entity"])
39     )
40
41     column = DatabaseUtils.get_column_represent_deleted(columns)
42     if column is not None:
43         if not DatabaseUtils.should_apply_filter(query, column):
44             return query
45
46     query = query.enable_assertions(False).where(column == None)
47
48     return query
```

Figure 5.31: File *src/modules/infrastructure/database/soft_delete_filter.py*

To satisfy the item 8 it was necessary to build a method that would build the *Query* to be executed according to the parameters received by the upper layers, as can be seen in the Figure 5.32.

Therefore it can be observed and concluded that the generic repository layer is capable of serving a wide range of *queries* making development easier.

Implementing the *soft delete* functionality was indeed a substantial endeavor, though it successfully fulfilled the outlined objective and offered transparency for developers to

```

30 def __apply_options(
31     self, query: Query, options_dict: Union[FindManyOptions, FindManyOptions] = None
32 ) -> Query:
33     if not options_dict:
34         return query
35
36     options_dict = self.__fix_options_dict(options_dict)
37     query = query.enable_assertions(False)
38
39     for key in options_dict.keys():
40         if key == "select":
41             query = query.options(load_only(*options_dict[key]))
42         elif key == "where":
43             query = query.where(*options_dict[key])
44         elif key == "order_by":
45             query = query.order_by(*options_dict[key])
46         elif key == "skip":
47             query = query.offset(options_dict[key])
48         elif key == "take":
49             query = query.limit(options_dict[key])
50         elif key == "relations":
51             query = query.options(subqueryload(getattr(self.entity, *options_dict[key])))
52         elif key == "with_deleted":
53             self.with_deleted = options_dict[key]
54         else:
55             raise KeyError(f"Unknown option: {key} in FindOptions")
56
57     return query
58

```

Figure 5.32: Method `__apply_options` inside the file `.../database/base_repository.py`

facilitate their work. However, it's crucial to acknowledge the trade-off involved: the additional overhead introduced by this feature can impact database performance. The overhead is primarily due to the extra resources required to manage the flagged 'deleted' records, which remain in the database. This can lead to increased storage use and potential slowdowns during query execution, particularly for large databases.

5.3 Recommendation System Development

In this section, we will provide a detailed description of the development of the recommendation system for the software, and how each step of the modeling process outlined in Section 4.6 of Chapter 4 was constructed to achieve the proposed goals.

We first divided the model into two major steps:

1. Ranking the foods based on each user's preferences and consumption history;
2. Properly filling the meals of the nutritional plan with the items containing the ranked foods from the previous step.

* Note: The restrictions set by the nutritionist will be considered only after all foods

have been scored. This means that restricted foods are not immediately excluded. The reason for this approach is that it's essential to collect data related to the specified foods in order to assign a score. Excluding them prior to this step could potentially lead to data loss, which is avoided by incorporating this sequence in our process.

With this division, we were able to collect only the necessary data for each step, as the output of step 1 is one of the required inputs for executing step 2. This also allowed us to create two endpoints, as shown in the image below (5.33).

```

23  @rs_router.post(
24      "/complete-nutritional-plan",
25      response_model=List[SimplifiedUserPreferencesTable],
26      dependencies=[Depends(Auth([UserRole.ADMIN, UserRole.NUTRITIONIST]))],
27  )
28  async def complete_nutritional_plan(
29      user_id: UUID,
30      nutritional_plan_id: UUID,
31      available: bool = True,
32      force_reload: bool = False,
33      database: Session = Depends(get_db),
34  ) -> Optional[List[SimplifiedUserPreferencesTable]]:
35      return await rs_service.complete_nutritional_plan(
36          str(user_id), str(nutritional_plan_id), available, force_reload, database
37      )
38
39
40  @rs_router.get(
41      "/food-preferences",
42      response_model=List[SimplifiedUserPreferencesTable],
43      dependencies=[Depends(Auth([UserRole.ADMIN, UserRole.NUTRITIONIST]))],
44  )
45  async def user_food_preferences(
46      user_id: UUID,
47      nutritional_plan_id: UUID,
48      available: bool = True,
49      force_reload: bool = False,
50      database: Session = Depends(get_db),
51  ) -> Optional[List[DetailedUserPreferencesTable]]:
52      return await rs_service.get_user_food_preferences(
53          str(user_id), str(nutritional_plan_id), available, force_reload, database
54      )

```

Figure 5.33: File `.../recommendation_system/controllers/recommendation_system_controller.py`

5.3.1 Food Ranking

To begin with, we will detail how the ranking process of all foods in the system was carried out, which variables were analyzed, and the procedure followed to obtain the response of

the *user_food_preferences* endpoint on line 45 of the Figure 5.33.

To rank foods based on preferences, we analyzed the foods that the user had specified as preferred/liked and those that were specified as not preferred/disliked.

The process is carried out for all foods that belong to the same deepest category as the food specified by the user, and the score of each food in the table is modified according to its degree of similarity to the specified food, as shown in Figure 5.34.

```
156     @staticmethod
157     def __define_score_by_preference_type(
158         foods_dt: pd.DataFrame,
159         food: pd.DataFrame,
160         food_category: pd.DataFrame,
161         foods_to_grade: List[Food],
162         positive: bool = True,
163     ) -> None:
164         for food_to_grade in foods_to_grade:
165             if food_to_grade.id == food["id"].iloc[0]:
166                 score = 50
167             elif food_to_grade.food_category_id == food_category["id"].iloc[0]:
168                 score = 25
169             elif food_to_grade.food_category.parent.id == food_category["parent"].iloc[0].id:
170                 score = 12
171             else:
172                 score = 7
173
174             if not positive:
175                 score *= -1
176
177         FindUserFoodPreferencesInterface.__award_score(foods_dt, food_to_grade.id, score)
```

Figure 5.34: File *.../interfaces/find_user_food_preferences_interface.py*

The scores assigned to each food, as shown in Figure 5.34 above, follow the following criteria:

- The highest score, in this case 50, on line 166, is assigned to the specified food
- Foods that belong to the same category as the specified food receive the second-highest score, in this case 25 on line 168, as they are closer to the specified food;
- Foods that are one level of distance away from the specified food receive an intermediate score, in this case 12 on line 170, as they aren't too far from the specified food but already show some degree of difference;

- Lastly, all other foods receive the lowest score, in this case 7 on line 172, as although they have some degree of similarity with the specified food, they present significant differences.

It is important to note that if the preference we are analyzing is of the type that the user does not like, the score assigned to that food will be negative, and therefore lines 174-175 exist.

Finally, we delve into notes related to the user's consumption pattern, analyzing all the foods consumed within specific time periods. The code for ranking these foods can be observed in the Figure 5.35 below, along with a detailed explanation of the logic developed.

```

179     def __award_score_by_consumption(
180         self, db_session: Session, user_id: str, foods_dt: pd.DataFrame
181     ) -> None:
182         fatigued_food = self.rs_repository.get_fatigued_food_from_user(
183             db_session, user_id, self.PERIOD_TO_FATIGUE, self.AMOUNT_TO_FATIGUE
184         )
185         user_consumption = self.rs_repository.get_user_consumption_last_days(
186             db_session, user_id, fatigued_food, self.PERIOD_TO_ANALYZE
187         )
188
189         for food_id in fatigued_food:
190             FindUserFoodPreferencesInterface.__award_score(foods_dt, food_id, -100)
191
192         for food_id, food_amount in user_consumption:
193             """It sets the score based on the amount consumed, if it's close to the amount of fatigue it receives
194             fewer points to discourage its recommendation.
195             If it's a little above the defined minimum amount it's understood that the user has some degree of
196             identification with that food.
197             And finally if it's below the minimum amount it's not possible to assume whether the user likes it
198             or not, so a lower score is assigned. Lower than the balance point but higher than that of the foods
199             close to fatigue, the intention is that this food is recommended to find out in the future whether the
200             user likes it or not."""
201             score = (
202                 10
203                 if food_amount >= floor(self.AMOUNT_TO_FATIGUE * self.PERCENTAGE_NEAR_FATIGUE)
204                 else 30
205                 if food_amount >= floor(self.AMOUNT_TO_FATIGUE * self.PERCENTAGE_FOR_IDENTIFICATION)
206                 else 20
207             )
208
209             FindUserFoodPreferencesInterface.__award_score(foods_dt, food_id, score)

```

Figure 5.35: File `.../interfaces/find_user_food_preferences_interface.py`

For a more detailed explanation, we will explain the step-by-step of the code developed in the following topics:

1. In lines *182-184*, all foods that the user is already fatigued from consuming are extracted. For this, a time period to be analyzed is defined by the variable *PERIOD_TO_FATIGUE*, and the quantity that must be consumed of the same food within this period is defined by the variable *AMOUNT_TO_FATIGUE*;
2. In lines *185-187*, all foods that the user consumed within a specific period, defined by the variable *PERIOD_TO_ANALYZE*, and that are not already fatigued foods, collected in item 1, are collected;
3. In lines *189-191*, foods that were considered fatigued due to the quantity that the user consumed, collected in item 1, are penalized with *-100 points*;
4. Finally, in lines *192-209*, scores are assigned to the foods collected in item 2, and the score assigned to them varies based on three criteria, which are:
 - A percentage considered near fatigue is defined by the variable *PERCENTAGE_NEAR_FATIGUE*. Therefore, if the analyzed food has been consumed enough to be considered close to fatigue, the score assigned is lower, only *10 points*, in order to avoid recommending this food too frequently.
 - A minimum percentage of food consumption is defined for the system to understand that the user identifies with this food, defined by the variable *PERCENTAGE_FOR_IDENTIFICATION*. If the analyzed food has been consumed enough to satisfy this condition and less than the amount to be considered close to fatigue, *30 points* are assigned.
 - Finally, if the analyzed food did not meet any of the previous requirements, only *20 points* are assigned to it, to try to recommend it more often, as the user has already consumed it, but it is not known yet whether they like it or not.

And thus, all the foods in the system are ranked. However, there is one last crucial step before the final response is delivered, which involves eliminating certain foods, if

required. This exclusion pertains to foods that have been previously marked as off-limits, either due to the user's allergies or because they are not allowed as per the user's current nutritional plan. It's important to understand that these restrictions are set by the nutritionist during the initial consultation and must be adhered to for all subsequent plan generations. Therefore, honoring these restrictions is an essential part of our algorithm before providing the final ranked list of foods.

After that, the ranked list of foods is returned, either for the next step which is the actual filling of the nutritional plan or for the endpoint that requested it.

It was decided to provide an endpoint for this step to assist the nutritionist in manually filling out the patient's nutritional plan if they prefer.

5.3.2 Filling Out the Nutritional Plan

Before completing the nutritional plan, in addition to the previous step, it is necessary to collect a few other pieces of information:

- Collect the types of meals included in the nutritional plan. Therefore, at least one day of the nutritional plan needs to be manually filled out by the nutritionist.
- With the output from the previous step, we have the permitted foods that can be consumed and ranked. However, meal recommendations are based on items that may consist of one or more foods. Therefore, only items that contain permitted foods are collected.
 - After collecting all permitted items for consumption, a score is assigned to each item by summing up the score of the individual foods that compose it.

After collecting this data, it is necessary to normalize it. Two normalizations are performed, which can be observed in the Figure 5.36 below:

- On *line 162*, a mitigation is performed on the percentage of permitted calories for consumption in meals if the sum exceeds 100% due to the number of meals in the nutritional plan.

- On *line 163*, the permissible amount of daily caloric intake is calculated based on the data provided in the nutritional plan. This calculation is crucial as it forms the basis for determining the quantities of various foods in the plan. It's worth noting that if there's a surplus in calorie count, the system will adjust the quantities of the foods accordingly. This automated adjustment ensures that the nutritional plan remains balanced and within the specified calorie range, thereby aiding in the effective management of the user's dietary requirements.

After collecting and normalizing the remaining data, the system is ready to make recommendations as necessary for days on the nutritional plan with no previously registered foods.

The food recommendations themselves take place in *lines 183-185* of the Figure 5.36 above, which invokes the `__suggest_meals` method that we can observe in the Figure 5.37 below.

The logic implemented in the `__suggest_meals` method, shown in Figure 5.37, for recommending meals is as follows:

- In *line 238*, the meal items to be recommended are sorted in descending order of their *score*.
- In *lines 241-245*, the items are divided into three groups based on the quantity size defined in *line 239*. These groups are:
 1. The first group includes the top 10% items with the highest scores, which are the ones with the highest likelihood of being consumed by the user.
 2. The second group comprises items that fall between the top 10% and 30% highest scores. These items may not be the user's preferred choices, but still have a considerable chance of being selected.
 3. The third and last group includes items with scores between 30% and 60%. These items have the lowest likelihood of being selected by the user.

```

152     async def __complete_nutritional_plan(
153         self,
154         nutritional_plan: NutritionalPlan,
155         user_item_preference: pd.DataFrame,
156         meal_plan: List[dict],
157         db: Session,
158     ) -> None:
159         nutrients = ["proteins", "lipids", "carbohydrates", "calories"]
160
161         date_list = self.__get_date_range(nutritional_plan)
162         adapted_meal_plan = self.__adapt_nutritional_values(meal_plan)
163         maximum_calories_per_day = self.__maximum_allowed_to_consume_per_day(
164             nutritional_plan, adapted_meal_plan, nutrients
165         )
166
167         for meal in adapted_meal_plan:
168             meal_items = self.__get_items_by_type_of_meal(meal, user_item_preference)
169
170             maximum_calories_in_meal = self.__get_maximum_calories_in_meal(
171                 maximum_calories_per_day, meal, nutrients
172             )
173             for meal_date in date_list:
174                 try:
175                     await self.nphm_interface.get_nutritional_plan_has_meal_by_date(
176                         meal_date, nutritional_plan.id, meal["meals_of_plan_id"], db
177                     )
178                 except NotFoundException:
179                     with keep_nested_transaction(db):
180                         new_nphm = await self.nphm_interface.create_nutritional_plan_has_meal(
181                             meal_date, nutritional_plan.id, meal["meals_of_plan_id"], db
182                         )
183                         await self.__suggest_meals(
184                             new_nphm.id, meal_items, maximum_calories_in_meal, db
185                         )
186                     db.commit()

```

Figure 5.36: File `.../interfaces/complete_nutritional_plan_interface.py`

- In *line 247*, a loop is executed for each of the three groups formed in the previous step, and the following task is performed:
 - In *lines 251-252*, a meal item is randomly selected to be recommended;
 - In *lines 253-255*, an ideal quantity of the selected item is determined based on

the nutritional limits established for the analyzed patient;

- Finally, in *lines 256-258*, the recommended item is inserted into the database.

```
231     async def __suggest_meals(  
232         self,  
233         nphm_id: UUID,  
234         meal_items: pd.DataFrame,  
235         maximum_calories_in_meal: dict,  
236         db: Session,  
237     ) -> None:  
238         meal_items = meal_items.sort_values("score", ascending=False)  
239         size = len(meal_items)  
240  
241         splitted_meals = [  
242             meal_items[0 : ceil(size * 0.1)],  
243             meal_items[ceil(size * 0.1) : ceil(size * 0.3)],  
244             meal_items[ceil(size * 0.3) : ceil(size * 0.6)],  
245         ]  
246  
247         for splitted_item in splitted_meals:  
248             if not len(splitted_item):  
249                 continue  
250  
251             index = randint(0, len(splitted_item) - 1)  
252             item = splitted_item.iloc[index]  
253             amount = CompleteNutritionalPlanInterface.__find_ideal_quantity(  
254                 item, maximum_calories_in_meal  
255             )  
256             await self.meals_options_interface.create_meal_option(  
257                 amount, True, item["id"], nphm_id, db  
258             )
```

Figure 5.37: File `.../interfaces/complete_nutritional_plan_interface.py`

5.4 Conclusion

In this chapter, we delved into the crucial elements of backend development for the software, focusing particularly on error handling, implementation of soft delete, and the development of the recommendation system.

Error handling was tactically addressed to provide a smooth and seamless user experience. Even when issues arise, the mechanisms in place ensure the user's interaction with the software is as undisturbed as possible.

The introduction of soft delete and the development of BaseRepository were pivotal elements for data maintenance and management. Soft delete allows for data preservation

even after deletion, which is vital for maintaining data integrity and providing ongoing insights into user behavior. The BaseRepository, on the other hand, serves as a common resource for handling database operations, making the code more efficient and easier to manage.

The development of the recommendation system was also a significant part of this chapter. A two-step process was outlined to rank foods based on the user's preferences and consumption history, and then fill the nutritional plan meals with items containing the ranked foods. Notably, constraints set by the nutritionist are considered only after all foods have been scored, thus avoiding any potential data loss.

This chapter underscored the complexity of backend development and the importance of detailed consideration of user needs and preferences when creating efficient functionalities. With the implementation of soft delete, BaseRepository, and the recommendation system, the backend provides a solid and flexible foundation for the software.

Chapter 6

Tests, Evaluation and Discussion

This chapter presents the tests performed as well as the building of the testing structure for the nutritional plan management and the recommendation system along with their respective results.

This chapter will also discuss the strengths of this work that we believe exceeded the initial project proposal, as well as the goals that couldn't be completed.

The most used frameworks for testing software development in Python are *unittest* and *pytest*, and the latter was chosen for this project, because it produces less test code which facilitates the development and maintenance among other benefits. This decision was anchored by the article [50] which makes a comparison between the two frameworks and why large projects are migrating from *unittest* to *Pytest*.

Still on *pytest*, the framework brings together some interesting features such as its flexibility and its differentiation from *fixtures*, the online article [51] provides a succinct explanation of these features being a great starting point to understand this tool.

6.1 Nutritional Management System Test Development and Results

For the software tests it was chosen, in the first stage, to perform only API tests in order to validate multiple scenarios and ensure that the content of the return *Json* is correct.

To make this possible it was necessary to create a database for testing and to ensure that changing the data in tests of other modules wouldn't impact the tests of another service, so it was necessary to somehow define the data and reload it to the database at every contextual change.

For this purpose, a service was developed that loads data from a *json* to the database in the way it was defined, see Figure 6.1, being able to load all tables of the system at once or only tables specified by the developer, passed by the parameter *entities_input* in the method *reload_fixture*, as it is possible to see the code in Figure 6.2.

After that we configured the *pytest* and specified that the default scope to be executed would be the *module* and therefore the loading of the fixtures would be executed at each change of *module* as can be seen in the method *__run_around_tests* of the Figure 6.3.

We took the opportunity to define some fixtures that would be used throughout the system that are the login credentials of users with different roles in the system, one of them can be observed in the method *user_common* also present in the Figure 6.3.

Before developing the tests, a class containing generic tests was developed that, if not all, most of the system services would have to execute and expect the same output, being possible to observe in the Figure 6.4 some of the methods developed.

Finally we develop the API tests, look at some examples through the Figure 6.5 and 6.6. Two points are important to note, which are:

- Each class only runs tests of a single service (endpoint) of that module, so a test class is needed for each service in the module;
- For each service in the module, tests are made in success and failure scenarios, because it's necessary to ensure that not only is the system working as expected,

```

1  [
2    {
3      "id": "950d760f-ba5c-44ca-b4ec-313510e59beb",
4      "created_at": "2022-07-26 15:57:40.502594 +00:00",
5      "updated_at": "2022-07-26 15:57:40.502594 +00:00",
6      "deleted_at": null,
7      "description": "Food 1",
8      "proteins": 0,
9      "lipids": 0,
10     "carbohydrates": 0,
11     "energy_value": 100,
12     "potassium": 0,
13     "phosphorus": 0,
14     "sodium": 0,
15     "food_category_id": "75827c83-d4cb-46cb-a092-9ba2dd962023"
16   },
17   {
18     "id": "e3ff57d6-eb77-48de-bb49-ff9201d95926",
19     "created_at": "2022-07-26 15:58:40.502594 +00:00",
20     "updated_at": "2022-07-26 15:58:40.502594 +00:00",
21     "deleted_at": null,
22     "description": "Food 2",
23     "proteins": 0,
24     "lipids": 0,
25     "carbohydrates": 0,
26     "energy_value": 95,
27     "potassium": 0,
28     "phosphorus": 0,
29     "sodium": 0,
30     "food_category_id": "90e719b0-0f32-4236-82e7-033e2deae8fd"
31   }
32 ]

```

Figure 6.1: File *test/fixtures/food.json*

but it must also be able to block and stop unexpected behavior and prevent data leakage.

6.1.1 Collected Results

In the Figure 6.7 you can see that 220 tests have run successfully, that is *100%* of the system tests have run successfully.

With this result it's possible to infer that in the scenarios in which the tests were developed the system is working as it should.

However to go further, using the *pytest-cov* plugin, which when running the tests produces code coverage reports, it was possible to analyze in the report that *96%* system coverage was achieved, as noted by the Figure 6.8.

```

45     @staticmethod
46     def get_entities() -> List[Table]:
47         return Base.metadata.sorted_tables
48
49     async def reload_fixtures(self, entities_input: List[DeclarativeMeta] = None) -> None:
50         entities = self.get_entities()
51
52         if entities_input:
53             apply_entities = []
54             for entity_input in entities_input:
55                 filtered_entity = list(
56                     filter(lambda entity: entity.name == entity_input.__tablename__, entities)
57                 )
58                 if not filtered_entity.__len__():
59                     raise Exception(f"Entity {entity_input.__name__} not found")
60                 else:
61                     apply_entities.append(filtered_entity[0])
62             entities = apply_entities
63
64             await self.clean_all(list(reversed(entities)))
65             await self.load_all(entities)
66
67     @staticmethod
68     async def clean_all(entities: List[Table]) -> None:
69         try:
70             for entity in entities:
71                 engine.execute(entity.delete())
72         except Exception as e:
73             raise Exception(f"Error cleaning test database: {e}")
74
75     async def load_all(self, entities: List[Table]) -> None:
76         try:
77             for entity in entities:
78                 items = self.__get_items_from_fixture(entity)
79                 if items.__len__() > 0:
80                     engine.execute(entity.insert(values=items))
81         except Exception as e:
82             raise Exception(f"Error loading fixtures on test database: {e}")

```

Figure 6.2: File *test/utils/database__config__test__utils.py*

This report also shows in detail the code coverage for each file in the system, and by analyzing the files that have the lowest coverage rate the following points can be highlighted:

- The entities codes that are not covered are from modules that don't have any services yet and therefore it isn't possible to reach these lines of code;
- The rest of the code that does not have full coverage is code that is either difficult to access in order to create an API test or is not achievable with this type of test and requires unit test;
- With the code coverage report it's possible to analyze the code that isn't being covered and develop better code to test it, but it's also possible to analyze code

```

18 # ----- PRIVATE METHODS -----
19 async def __login_user(
20     user_email: str, user_password: str = "12345678"
21 ) -> Optional[LoginPayloadDto]:
22     return await auth_service.login_user(
23         OAuth2PasswordRequestForm(username=user_email, password=user_password, scope=""),
24         db_test_utils.db,
25     )
26
27
28 # ----- PRIVATE FIXTURES -----
29 @pytest.fixture(scope="module", autouse=True)
30 async def __run_around_tests(request: FixtureRequest) -> None:
31     fixtures_to_reload = (
32         request.module.fixtures_to_reload if hasattr(request.module, "fixtures_to_reload") else None
33     )
34
35     await db_test_utils.reload_fixtures(fixtures_to_reload)
36
37     yield
38
39     await db_test_utils.close_db_connection()
40
41
42 # ----- PUBLIC FIXTURES -----
43 @pytest.fixture(scope="module")
44 def event_loop() -> asyncio.AbstractEventLoop:
45     return asyncio.get_event_loop()
46
47
48 @pytest.fixture(scope="module")
49 async def user_common() -> Optional[LoginPayloadDto]:
50     user_common = db_test_utils.get_entity_objects(User)[0]
51     return await __login_user(user_common["email"])
52

```

Figure 6.3: File *confest.py*

that is no longer used and can be removed from the system.

Another test performed was to measure the response time of each endpoint of the system in order to measure the average response time, but also to analyze which codes is already performing well and which need improvements, and to analyze if there is any connection between them. The result can be seen by the graph in the Figure 6.9.

To provide a more detailed visualization of the response times of the endpoints, treemaps graphs have been generated. The one generated for the *Food* module can be seen in Figure 6.10.

Through the Figure 6.10 notice that the test case that took the longest time to execute was the *Create New Food* and observing the only piece of code that this endpoint executes, see Figure 6.11, we notice that there is a check of the relationships that are being inserted if they are valid. This is an overhead generated by the *soft delete*, and taking into account

```

9  class TestBaseE2E:
10     db_test_utils = DatabaseConfigTest()
11
12     base_url = "http://localhost:3000"
13
14     # ----- PUBLIC METHODS -----
15     async def get_no_authentication(self, route: str) -> None:
16         async with AsyncClient(app=app, base_url=self.base_url) as ac:
17             assert (await ac.get(route)).status_code == HTTP_401_UNAUTHORIZED
18
19     async def get_different_required_authentication(
20         self, route: str, login_payload: LoginPayloadDto
21     ) -> None:
22         async with AsyncClient(app=app, base_url=self.base_url) as ac:
23             assert (
24                 await ac.get(
25                     route,
26                     headers={"Authorization": f"Bearer {login_payload.access_token}"},
27                 )
28             ).status_code == HTTP_403_FORBIDDEN
29
30     async def del_no_authentication(self, route: str) -> None:
31         async with AsyncClient(app=app, base_url=self.base_url) as ac:
32             assert (await ac.delete(route)).status_code == HTTP_401_UNAUTHORIZED
33
34     async def del_different_required_authentication(
35         self, route: str, login_payload: LoginPayloadDto
36     ) -> None:
37         async with AsyncClient(app=app, base_url=self.base_url) as ac:
38             assert (
39                 await ac.delete(
40                     route,
41                     headers={"Authorization": f"Bearer {login_payload.access_token}"},
42                 )
43             ).status_code == HTTP_403_FORBIDDEN

```

Figure 6.4: File *test/test_base_e2e.py*

that the entity *Food* is one of those that has the largest number of relationships in the system it's possible to justify therefore that this would be the cause for the high response time.

Also, something that must be taken into consideration is that the tests are being run in a development environment and therefore the amount of data in the database is minimal, and that in a production environment with a fully populated database it's expected that the most time demanding endpoints will be of the type "GET".

So with the tests that have been performed on the system and their respective results,

```

24 CONTROLLER = "food"
25 food_service = FoodService()
26
27
28 @pytest.mark.describe(f"POST Route: /{CONTROLLER}/create")
29 class TestCreateFood(TestBaseE2E):
30     route = f"/{CONTROLLER}/create"
31
32     @pytest.mark.asyncio
33     @pytest.mark.it("Success: Create a food")
34     async def test_create_new_food(self, user_admin: Optional[LoginPayloadDto]) -> None:
35         async with AsyncClient(app=app, base_url=self.base_url) as ac:
36             response = await ac.post(
37                 self.route,
38                 json={
39                     "description": "Food Test 1",
40                     "proteins": 0,
41                     "lipids": 0,
42                     "carbohydrates": 0,
43                     "energy_value": 95,
44                     "potassium": 0,
45                     "phosphorus": 0,
46                     "sodium": 0,
47                     "food_category": "42853fe7-5bf7-4503-af2c-2b284b5fdfbe",
48                 },
49                 headers={"Authorization": f"Bearer {user_admin.access_token}"},
50             )
51
52             data = response.json()
53
54             assert response.status_code == HTTP_201_CREATED
55             assert [hasattr(data, attr_name) for attr_name in ["id", "energy_value"]]
56             assert data["energy_value"] == 95
57
58             activity_level_dto = await food_service.find_one_food(data["id"], self.db_test_utils.db)
59
60             assert activity_level_dto is not None
61             assert activity_level_dto.energy_value == data["energy_value"]

```

Figure 6.5: File *src/modules/domain/food/tests/test_food_e2e.py*

it is possible to infer that the system is working correctly in the scenario in which it was imagined and any changes can be easily detected.

Up until this point, the API tests developed have managed to cover a significant portion of the system. This provides assurance that the tests' efficiency is at an acceptable level for this stage of the project.

While it would be desirable to incorporate other forms of testing, such as unit tests, to increase software security and enable continuous integration (CI) and continuous delivery (CD), these are not strictly necessary for the project's initial phase.

With the performance tests it was possible to observe which endpoints have satisfactory response times and which need optimization, and to verify that although the median is

```

139 @pytest.mark.describe(f"GET Route: /{CONTROLLER}/get")
140 class TestGetAllFoods(TestBaseE2E):
141     route = f"/{CONTROLLER}/get"
142
143     @pytest.mark.asyncio
144     @pytest.mark.it("Success: Get a list of all food")
145     async def test_get_foods(self, user_admin: Optional[LoginPayloadDto]) -> None:
146         async with AsyncClient(app=app, base_url=self.base_url) as ac:
147             response = await ac.get(
148                 self.route,
149                 headers={"Authorization": f"Bearer {user_admin.access_token}"},
150                 params={"columns": ["description", "food_category"]},
151             )
152
153             body = response.json()
154             data = body["data"]
155
156             assert response.status_code == HTTP_200_OK
157             assert body["count"] >= 0
158
159             for food_category in data:
160                 if food_category["id"] == "950d760f-ba5c-44ca-b4ec-313510e59beb":
161                     assert food_category["description"] == "Food 1"
162                     assert (
163                         food_category["food_category"]["id"] == "75827c83-d4cb-46cb-a092-9ba2dd962023"
164                     )
165                     assert food_category["food_category"]["food_category"] is None
166
167     @pytest.mark.asyncio
168     @pytest.mark.it("Failure: Get a list of all food without authentication")
169     async def test_no_authentication(self) -> None:
170         await self.get_no_authentication(self.route)
171
172     @pytest.mark.asyncio
173     @pytest.mark.it("Failure: Get a list of all food with non required authentication")
174     @pytest.mark.parametrize("user", ["user_common"])
175     async def test_different_required_authentication(
176         self, user: str, request: FixtureRequest
177     ) -> None:
178         user: LoginPayloadDto = request.getfixturevalue(user)
179         await self.get_different_required_authentication(self.route, user)

```

Figure 6.6: File `src/modules/domain/food/tests/test_food_e2e.py`

```

PATCH Route: /user/update/<id>
✓ Success: Update user nutritionist
✓ Success: Update user with deleted user email
✓ Failure: Update user with email that already exists
✓ Failure: Update deleted user
✓ Failure: Update user without authentication
[100%]

220 passed in 7.89s

```

Figure 6.7: Test success rate

less than 20 ms, the overheads generated by the *soft delete* impact the system as a whole, generating outliers of up to 100 ms, which represents an increase of 500% compared to

Coverage report: 96%

Module	statements	missing	excluded	coverage %
src/modules/domain/biochemical_data/entities/biochemical_data_entity.py	48	20	0	58%
src/core/utls/image_utls.py	40	15	0	62%
src/modules/domain/specificity/services/specificity_type_service.py	19	7	0	63%
src/modules/app/app_service.py	3	1	0	67%
src/core/constants/enum/periods.py	13	4	0	69%
src/core/utls/pagination_utls.py	108	31	0	71%
src/modules/domain/nutritional_plan/entities/nutritional_plan_entity.py	28	8	0	71%
src/modules/domain/meal/dto/food_can_eat_at/food_can_eat_at_dto.py	18	5	0	72%
src/modules/domain/diagnosis/entities/diagnosis_entity.py	19	5	0	74%
src/modules/domain/plan_meals/entities/meals_of_plan_entity.py	19	5	0	74%
src/modules/domain/meal/services/food_can_eat_at_service.py	16	4	0	75%
src/modules/domain/anthropometric_data/services/anthropometric_data_service.py	94	21	0	78%
src/modules/domain/specificity/entities/specificity_entity.py	18	4	0	78%
src/modules/domain/diary/entities/diary_entity.py	15	3	0	80%
src/modules/domain/meal/entities/food_can_eat_at_entity.py	15	3	0	80%
src/modules/infrastructure/auth/tests/test_auth_e2e.py	54	11	0	80%
test/utls/database_config_test_utls.py	59	12	0	80%
src/modules/domain/forbidden_foods/entities/forbidden_foods_entity.py	16	3	0	81%
src/modules/domain/plan_meals/dto/meals_of_plan/meals_of_plan_dto.py	17	3	0	82%
src/modules/domain/specificity/entities/specificity_type_entity.py	11	2	0	82%
src/modules/domain/appointment/services/appointment_service.py	61	10	0	84%
src/modules/infrastructure/user/user_service.py	87	14	0	84%
src/modules/app/app_controller.py	7	1	0	86%
src/modules/domain/nutritional_plan/dto/nutritional_plan_dto.py	22	3	0	86%
src/modules/domain/food/interfaces/food_interface.py	15	2	0	87%
src/core/middlewares/limit_upload_size.py	17	2	0	88%
src/modules/domain/antecedent/dto/antecedent/antecedent_dto.py	18	2	0	89%
src/core/utls/json_utls.py	10	1	0	90%

Figure 6.8: Code coverage report from *pytest-cov*

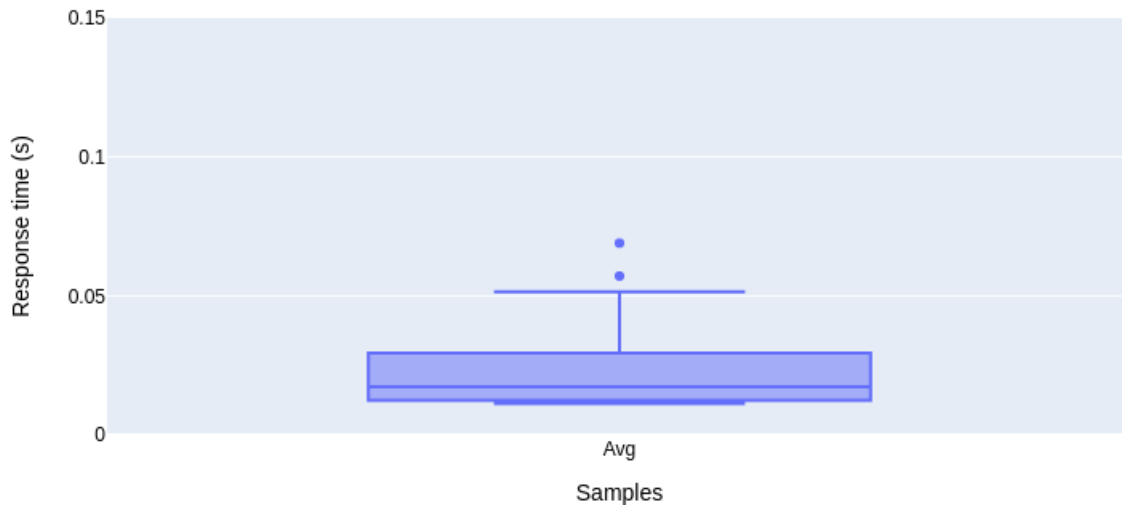


Figure 6.9: Response time distribution of the system endpoints



Figure 6.10: Food test response times

```

288     async def create(self, _entity: Union[T, BaseModel], db: Session = next(get_db())) -> T:
289         if isinstance(_entity, BaseModel):
290             partial_data_entity = _entity.dict(exclude_unset=True)
291             _entity = self.entity(**partial_data_entity)
292
293         await self.__is_relations_valid(db, _entity.__dict__)
294
295         db.add(_entity)
296         return _entity

```

Figure 6.11: *Base Repository* create method

the median.

6.2 Recommendation System Tests and Results

In order to ensure that the recommendation system methods developed behave as expected, endpoint tests were performed. These tests for this stage followed the same structure as the tests previously described in the previous section for the system as a whole, and the coverage and performance results from the previous section include the recommendation system tests.

The endpoint tests were designed to evaluate the functionality of the recommendation system, including its ability to suggest meals based on user preferences and nutritional needs. The tests were performed using a variety of input data and scenarios to ensure that the system was able to handle different situations and provide accurate recommendations.

In order to assess the accuracy of the food recommendations provided by the system

and determine if they met the user's nutritional needs and preferences, as well as offered diverse meal options, four nutritional plans were developed with varying amounts of data available for the system to analyze the user's profile. These included:

1. A nutritional plan in which only the daily caloric intake data was provided;
2. A nutritional plan with the caloric intake data and a few entries from the user's consumption history;
3. A nutritional plan with the caloric intake data and a few entries regarding the user's preferences;
4. A nutritional plan with the caloric intake data, a few entries of the user's preferences, and consumption history.

Before requesting the system to generate nutritional plans with the aim of analyzing its behavior, it was crucial to input meal information, as this data did not previously exist and needed to be created. To assist in this task, we used ChatGPT [52] to provide complete meals, including all the necessary ingredients. The prompts used to generate these meals can be observed below:

- **For heavy meals:** Act like a nutritionist and generate 5 healthy recipes for heavy meals, which must necessarily have a protein source, a carbohydrate source, a salad or vegetables, and finally a soup and a healthy drink to accompany them.
- **For light meals:** Act like a nutritionist and generate 5 healthy recipes for light meals, such as breakfast, mid-morning, afternoon snack and supper. These meals can contain fruit, cheese and/or dairy products among others...
- **For meals that can be consumed in light and heavy meals:** Act like a nutritionist and generate 5 healthy recipes that can be consumed in light meals that are breakfast, mid-morning, afternoon snack and supper. These meals can contain fruit, cheese and/or dairy products among others, but can also be consumed in

heavy meals that need to have a source of carbohydrates, a source of protein and some nutrient from a vegetable or salad.

- **Meals after waking up:** Act like a nutritionist and generate 10 healthy recipes that can be consumed in meals to be taken as soon as you wake up that constitute a minimum amount of liquids to be consumed

After the meals were generated and collected, manual processing was conducted, due to the small sample size, to translate the food items in each meal to those that belonged to the system's initial database. However, not all meals contained all the food items or similar ones registered in the database, and thus, some were not imported.

Following this, the system was requested to populate a nutritional plan for each of the scenarios described earlier in this section in sequence. A preview of the results, showcasing just one day from each scenario, can be observed through the images 6.12, 6.13, 6.14, 6.15. It is important to note that each row represents a different option aiming to achieve the caloric intake for that particular meal. To accomplish this, the portion size of the dish is varied, as represented by the *amount* column.

meal_date	meal_description	meal_start_time	meal_end_time	food_description	amount
2023-04-21	Ao acordar	09:00:00+00	09:30:00+00	Chá de hortelã	0,5
				Limonada com gengibre	0,5
				Suco de melancia com hortelã	0,5
	Pequeno-almoço	10:30:00+00	11:00:00+00	Sanduíche natural com queijo e peito de peru	0,5
				Biscoitos integrais com geleia de frutas sem açúcar e queijo	0,5
				Salada de frutas com iogurte e coco ralado	0,5
	Meio da Manhã	11:30:00+00	12:00:00+00	Sanduíche de frango desfiado com cenoura e alface	0,5
				Biscoitos integrais com geleia de frutas sem açúcar e queijo	0,5
				Smoothie de morango e banana	0,5
	Almoço	13:00:00+00	14:00:00+00	Sanduíche de pão sírio com pasta de grão-de-bico e legumes	1
				Omelete de legumes com macarrão de abobrinha e salada	0,5
				Bife de alcatra ao molho de cogumelos com arroz selvagem	0,5
	Lanche da tarde	16:00:00+00	16:30:00+00	Sanduíche integral com atum e salada	0,5
				Salada de grão-de-bico com atum e vegetais	0,5
				Biscoitos integrais com geleia de frutas sem açúcar e queijo	0,5
	Jantar	21:00:00+00	22:00:00+00	Tofu grelhado com purê de mandioquinha e brócolis no vapor	1
				Salada de grão-de-bico com atum e vegetais	1
				Torrada integral com queijo quark e frutas	2,5
	Ceia	23:30:00+00	00:00:00+00	Biscoitos integrais com geleia de frutas sem açúcar e queijo	0,5
				Sanduíche integral com atum e salada	0,5
				Bowl de frutas com chia e mel	0,5

Figure 6.12: Nutritional plan without any input

meal_date	meal_description	meal_start_time	meal_end_time	food_description	amount
2023-04-21	Ao acordar	09:00:00+00	09:30:00+00	Chá de hortelã	0,5
				Suco de melancia com hortelã	0,5
				Suco verde detox	0,5
	Pequeno-almoço	10:30:00+00	11:00:00+00	Pão com ovo e tomate	0,5
				Overnight oats com frutas	0,5
				Torradas com queijo cottage e geleia de frutas sem açúcar	0,5
	Meio da Manhã	11:30:00+00	12:00:00+00	Parfait de iogurte, frutas e chia	0,5
				Iogurte com frutas e granola	0,5
				Pão com ovo e abacate	0,5
	Almoço	13:00:00+00	14:00:00+00	Filé de peixe ao molho de alcaparras com purê de batata-c	1
				Risoto de cogumelos com lombo suíno grelhado e salada c	0,5
				Ovos mexidos com abacate e torrada integral	0,5
	Lanche da tarde	16:00:00+00	16:30:00+00	Sanduíche integral com atum e salada	0,5
				Parfait de iogurte, frutas e chia	0,5
				Iogurte com frutas e granola	0,5
	Jantar	21:00:00+00	22:00:00+00	Tortilha de batata-doce com espinafre e queijo	1
				Sanduíche de pão sírio com pasta de grão-de-bico e legum	1
				Estrogonofe de frango com arroz integral e salada de pepi	2,5
	Ceia	23:30:00+00	00:00:00+00	Overnight oats com frutas	0,5
				Smoothie de morango e banana	0,5
				Parfait de iogurte, frutas e chia	0,5

Figure 6.13: Nutritional plan with historical consumption data only

meal_date	meal_description	meal_start_time	meal_end_time	food_description	amount
2023-04-21	Ao acordar	09:00:00+00	09:30:00+00	Chá de hortelã	0,5
				Limonada com gengibre	0,5
				Suco verde detox	0,5
	Pequeno-almoço	10:30:00+00	11:00:00+00	Torrada integral com queijo quark e frutas	0,5
				Sanduíche natural com queijo e peito de peru	0,5
				Pão com ovo e tomate	0,5
	Meio da Manhã	11:30:00+00	12:00:00+00	Crepioca com queijo e espinafre	0,5
				Sanduíche natural com queijo e peito de peru	0,5
				Mingau de aveia com frutas secas e canela	0,5
	Almoço	13:00:00+00	14:00:00+00	Frango grelhado com batata-doce assada e salada de tom	1,5
				Filé mignon suíno com polenta e salada de espinafre e ma	0,5
				Estrogonofe de frango com arroz integral e salada de pepi	1
	Lanche da tarde	16:00:00+00	16:30:00+00	Pão com queijo cottage e tomate	0,5
				Pão com ovo e tomate	0,5
				Sanduíche natural com queijo e peito de peru	0,5
	Jantar	21:00:00+00	22:00:00+00	Risoto de cogumelos com lombo suíno grelhado e salada c	0,5
				Tortilha de batata-doce com espinafre e queijo	1,5
				Torrada integral com queijo quark e frutas	2,5
	Ceia	23:30:00+00	00:00:00+00	Sanduíche integral com peito de peru, queijo cottage e rú	0,5
				Mingau de aveia com frutas secas e canela	0,5
				Iogurte com frutas e granola	0,5

Figure 6.14: Nutritional plan with user preferences only

meal_date	meal_description	meal_start_time	meal_end_time	food_description	amount
2023-04-21	Ao acordar	09:00:00+00	09:30:00+00	Chá de hortelã	0,5
				Água aromatizada com pepino e limão	0,5
				Suco de melancia com hortelã	0,5
	Pequeno-almoço	10:30:00+00	11:00:00+00	Bowl de iogurte, aveia e frutas com amêndoas	0,5
				Overnight oats com frutas	0,5
				Torradas com pasta de grão-de-bico e tomate	0,5
	Meio da Manhã	11:30:00+00	12:00:00+00	Sanduíche integral com atum e salada	0,5
				Iogurte com frutas e granola	0,5
				Sanduíche integral com peito de peru, queijo cottage e rúcula	0,5
	Almoço	13:00:00+00	14:00:00+00	Frango assado com batata doce e salada de espinafre	1
				Filé mignon suíno com polenta e salada de espinafre e maionese	0,5
				Tortilha de batata-doce com espinafre e queijo	1,5
	Lanche da tarde	16:00:00+00	16:30:00+00	Pão com ovo e tomate	0,5
				Mingau de aveia com frutas secas e canela	0,5
				Bowl de iogurte, aveia e frutas com amêndoas	0,5
	Jantar	21:00:00+00	22:00:00+00	Sanduíche de pão sírio com pasta de grão-de-bico e legumes	1
				Estrogonofe de frango com arroz integral e salada de pepino	1
				Tortilha de batata-doce com espinafre e queijo	1,5
	Ceia	23:30:00+00	00:00:00+00	Mingau de aveia com frutas secas e canela	0,5
				Sanduíche natural com queijo e peito de peru	0,5
				Bowl de iogurte, aveia e frutas com amêndoas	0,5

Figure 6.15: Nutritional plan with user preferences and consumption history

It has been observed that even with limited data input and a small number of registered meals, the system is able to generate a wide variety of meal options to cater to diverse tastes. Furthermore, it is evident that the more user data is available in the system, the better it can adapt and improve the suggested meals. Consequently, it can be concluded that although the proposed recommendation system requires further refinement, it is already suitable for initial use.

6.3 Conclusion

In this work we seek to achieve the initial proposal of the project and we model the system and the RS in order to achieve this objective.

During the building of the structure for the system development, we developed several mechanisms to facilitate the implementation of the business rules of each service, which are:

- Easy organization of each module and a fast module development process;
- System-wide standard error throwing mechanism;

- Generic pagination system for all endpoints that allows a wide range of queries to the database;
- Code encapsulation that allows the creation of several steps of a module with a few lines, including the creation of new tables with soft delete enabled;
- Standardization of how request bodies should arrive at the server and responses to these requests to form good documentation that helps and speeds up front- and backend development;
- Insertion of security layers to block unexpected behavior to prevent data leakage and system misuse;
- Creation of a generic repository capable of supporting a wide range of database queries and with the implementation of the fully functional soft delete in all methods.

Although with all this structure in place that allowed us to create an entire module with a complete simple (Create, Read, Update, Delete) (CRUD) in less than 1 hour of development, it wasn't possible to finish all the services of the system because these required specific business rules that although they were detailed would depend on the backend and front-end working together.

So that the code developed on the backend was tested and validated by the front-end in order to avoid refactoring, unnecessary code development, and that the code produced could serve the front-end in the best possible way.

The recommendation system achieved favorable results in the testing environment, demonstrating its promise and readiness for use in a production setting. As more data is generated and becomes suitable for analysis, the system can be further refined and improved.

Chapter 7

Conclusion

The work described consisted in developing a system that would help nutritionists with their daily tasks, including filling out the nutritional plan with meal recommendations based on the patient's profile in conjunction with the nutritional plan, but also in increasing the efficiency of the nutritionist's work with detailed patient analysis.

The work was also aimed at enhancing the patient's experience by quickly and conveniently providing all the meals in the plan and their status. Furthermore, the backend was designed to enable the front-end to generate personalized performance reports. These reports, based on the feedback provided by patients throughout their journey of following the nutritional plan, are intended to motivate and encourage them to adhere to the plan.

During the research stage, we deepened our knowledge in several areas, especially regarding the techniques used in recommendation systems, such as the works [22], [23], [24], [25] and [26].

A mention goes to the works [31] and [32] that have provided some guidelines of elements that a nutritional assistant system should contain or be covered by.

With the modeling of the system and the details of how the development was done, it's possible to conclude that the system and the RS is well on its way not only to reaching the proposed goals but also to exceeding them with a good structure that allows the easy addition of new functionalities but also guarantees an excellent performance of those already contemplated by the system as a whole.

Lastly, the tests conducted in the development environment ensure that the development is progressing smoothly, with excellent results in terms of behavior, coverage, and response time of the endpoints. The outcomes obtained from the recommendation system have proven to be promising for this initial version, although further refinement is still needed.

7.1 Future Works

At this point, the process of integrating the front-end with the backend is underway, and the complete system is being deployed on a virtual machine at CeDRI. This will enable users, both nutritionists and patients, to conduct usability testing.

Considering the backend, for the next steps of the process it is necessary to continue developing the functionalities that were not possible to deliver in this first phase, but it is also necessary to add new functionalities as needed by the nutritionist.

It's necessary to collect and analyze the response times of the endpoints in the production environment to see if they correspond to those of the tests in the development environment or if there are any processing bottlenecks that can be improved.

Also, it's desirable to continue improving the *Base Repository* that was built in this work, in order to add support for new features and improve the performance of the existing ones, especially those related to the *soft delete*.

In conclusion, regarding the RS, our future goals include automating the creation and importation of new meals into the system's database. Additionally, as more data is generated within the system, we aim to apply machine learning and artificial intelligence algorithms to improve the metrics used in assigning scores to food items and meals. This will not only preserve the variety but also make the recommendations more accurate and better suited to the needs of both the patient and the nutritionist.

Bibliography

- [1] W. C. Willett and M. J. Stampfer, “Current evidence on healthy eating”, *Annual review of public health*, vol. 34, pp. 77–95, 2013.
- [2] L. Tian, B. Yang, X. Yin, and Y. Su, “A survey of personalized recommendation based on machine learning algorithms”, in *Proceedings of the 2020 4th International Conference on Electronic Information Technology and Computer Engineering*, ser. EITCE 2020, Xiamen, China: Association for Computing Machinery, 2020, pp. 602–610, ISBN: 9781450387811. DOI: 10 . 1145 / 3443467 . 3444711. [Online]. Available: <https://doi-org.ez48.periodicos.capes.gov.br/10.1145/3443467.3444711>.
- [3] Y. Ning, Y. Shi, L. Hong, H. Rangwala, and N. Ramakrishnan, “A gradient-based adaptive learning framework for efficient personal recommendation”, in *Proceedings of the Eleventh ACM Conference on Recommender Systems*, ser. RecSys ’17, Como, Italy: Association for Computing Machinery, 2017, pp. 23–31, ISBN: 9781450346528. DOI: 10 . 1145 / 3109859 . 3109909. [Online]. Available: <https://doi-org.ez48.periodicos.capes.gov.br/10.1145/3109859.3109909>.
- [4] S. P. Mudur, S. A. Mokhov, and Y. Mao, “A framework for enhancing deep learning based recommender systems with knowledge graphs”, in *25th International Database Engineering & Applications Symposium*, New York, NY, USA: Association for Computing Machinery, 2021, pp. 11–20, ISBN: 9781450389914. DOI: 10 . 1145 / 3472163 . 3472183. [Online]. Available: <https://doi-org.ez48.periodicos.capes.gov.br/10.1145/3472163.3472183>.

- [5] Q. Han, C. Zhang, R. Chen, R. Lai, H. Song, and L. Li, “Multi-faceted global item relation learning for session-based recommendation”, in *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '22, Madrid, Spain: Association for Computing Machinery, 2022, pp. 1705–1715, ISBN: 9781450387323. DOI: 10.1145/3477495.3532024. [Online]. Available: <https://doi-org.ez48.periodicos.capes.gov.br/10.1145/3477495.3532024>.
- [6] J. Guo, “Research on hybrid recommendation algorithm based on personalized learning resource recommendation model”, in *2021 3rd International Conference on Artificial Intelligence and Advanced Manufacture*, ser. AIAM2021, Manchester, United Kingdom: Association for Computing Machinery, 2021, pp. 727–732, ISBN: 9781450385046. DOI: 10.1145/3495018.3495148. [Online]. Available: <https://doi-org.ez48.periodicos.capes.gov.br/10.1145/3495018.3495148>.
- [7] Y. G. Cinar and J.-M. Renders, “Adaptive pointwise-pairwise learning-to-rank for content-based personalized recommendation”, ser. RecSys '20, Virtual Event, Brazil: Association for Computing Machinery, 2020, pp. 414–419, ISBN: 9781450375832. DOI: 10.1145/3383313.3412229. [Online]. Available: <https://doi-org.ez48.periodicos.capes.gov.br/10.1145/3383313.3412229>.
- [8] Y. Liu and J. Yang, “A novel learning-to-rank based hybrid method for book recommendation”, in *Proceedings of the International Conference on Web Intelligence*, ser. WI '17, Leipzig, Germany: Association for Computing Machinery, 2017, pp. 837–842, ISBN: 9781450349512. DOI: 10.1145/3106426.3106547. [Online]. Available: <https://doi-org.ez48.periodicos.capes.gov.br/10.1145/3106426.3106547>.
- [9] S. T. Jishan and Y. Wang, “Audience activity recommendation using stacked-lstm based sequence learning”, ser. ICMLC 2017, Singapore, Singapore: Association for Computing Machinery, 2017, pp. 98–106, ISBN: 9781450348171. DOI: 10.1145/3055635.3056606. [Online]. Available: <https://doi-org.ez48.periodicos.capes.gov.br/10.1145/3055635.3056606>.

- [10] F. Mi, X. Lin, and B. Faltings, “Ader: Adaptively distilled exemplar replay towards continual learning for session-based recommendation”, ser. RecSys ’20, Virtual Event, Brazil: Association for Computing Machinery, 2020, pp. 408–413, ISBN: 9781450375832. DOI: 10.1145/3383313.3412218. [Online]. Available: <https://doi-org.ez48.periodicos.capes.gov.br/10.1145/3383313.3412218>.
- [11] Z. Pan, F. Cai, Y. Ling, and M. de Rijke, “An intent-guided collaborative machine for session-based recommendation”, ser. SIGIR ’20, Virtual Event, China: Association for Computing Machinery, 2020, pp. 1833–1836, ISBN: 9781450380164. DOI: 10.1145/3397271.3401273. [Online]. Available: <https://doi-org.ez48.periodicos.capes.gov.br/10.1145/3397271.3401273>.
- [12] C. Su, Z. Hu, and X. Xie, “Cross-domain recommendation based on heterogeneous information network with adversarial learning”, ser. ISMSI 2021, Victoria, Seychelles: Association for Computing Machinery, 2021, pp. 65–70, ISBN: 9781450389679. DOI: 10.1145/3461598.3461609. [Online]. Available: <https://doi-org.ez48.periodicos.capes.gov.br/10.1145/3461598.3461609>.
- [13] K. Ong, S.-C. Haw, and K.-W. Ng, “Deep learning based-recommendation system: An overview on models, datasets, evaluation metrics, and future trends”, ser. CIIS 2019, Bangkok, Thailand: Association for Computing Machinery, 2019, pp. 6–11, ISBN: 9781450372596. DOI: 10.1145/3372422.3372444. [Online]. Available: <https://doi-org.ez48.periodicos.capes.gov.br/10.1145/3372422.3372444>.
- [14] S. Zhang, L. Yao, A. Sun, and Y. Tay, “Deep learning based recommender system: A survey and new perspectives”, *ACM Comput. Surv.*, vol. 52, no. 1, Feb. 2019, ISSN: 0360-0300. DOI: 10.1145/3285029. [Online]. Available: <https://doi-org.ez48.periodicos.capes.gov.br/10.1145/3285029>.
- [15] R. Guan, H. Pang, F. Giunchiglia, X. Li, X. Yang, and X. Feng, “Deployable and continuable meta-learning-based recommender system with fast user-incremental updates”, ser. SIGIR ’22, Madrid, Spain: Association for Computing Machinery,

- 2022, pp. 1423–1433, ISBN: 9781450387323. DOI: 10.1145/3477495.3531964. [Online]. Available: <https://doi-org.ez48.periodicos.capes.gov.br/10.1145/3477495.3531964>.
- [16] Q. Zhao, F. M. Harper, G. Adomavicius, and J. A. Konstan, “Explicit or implicit feedback? engagement or satisfaction? a field experiment on machine-learning-based recommender systems”, ser. SAC ’18, Pau, France: Association for Computing Machinery, 2018, pp. 1331–1340, ISBN: 9781450351911. DOI: 10.1145/3167132.3167275. [Online]. Available: <https://doi-org.ez48.periodicos.capes.gov.br/10.1145/3167132.3167275>.
- [17] T. Silva, N. Silva, H. Werneck, A. C. M. Pereira, and L. Rocha, “The impact of first recommendations based on exploration or exploitation approaches in recommender systems’ learning”, in *Proceedings of the Brazilian Symposium on Multimedia and the Web*, ser. WebMedia ’20, São Luis, Brazil: Association for Computing Machinery, 2020, pp. 173–180, ISBN: 9781450381963. DOI: 10.1145/3428658.3430971. [Online]. Available: <https://doi-org.ez48.periodicos.capes.gov.br/10.1145/3428658.3430971>.
- [18] W. Meng, D. Yang, and Y. Xiao, “Incorporating user micro-behaviors and item knowledge into multi-task learning for session-based recommendation”, ser. SIGIR ’20, Virtual Event, China: Association for Computing Machinery, 2020, pp. 1091–1100, ISBN: 9781450380164. DOI: 10.1145/3397271.3401098. [Online]. Available: <https://doi-org.ez48.periodicos.capes.gov.br/10.1145/3397271.3401098>.
- [19] J. Jin, X. Chen, W. Zhang, J. Huang, Z. Feng, and Y. Yu, “Learn over past, evolve for future: Search-based time-aware recommendation with sequential behavior data”, in *Proceedings of the ACM Web Conference 2022*, ser. WWW ’22, Virtual Event, Lyon, France: Association for Computing Machinery, 2022, pp. 2451–2461, ISBN: 9781450390965. DOI: 10.1145/3485447.3512117. [Online]. Available: <https://doi-org.ez48.periodicos.capes.gov.br/10.1145/3485447.3512117>.

- [20] Y. Li, W. Chen, and H. Yan, “Learning graph-based embedding for time-aware product recommendation”, in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, ser. CIKM '17, Singapore, Singapore: Association for Computing Machinery, 2017, pp. 2163–2166, ISBN: 9781450349185. DOI: 10.1145/3132847.3133060. [Online]. Available: <https://doi-org.ez48.periodicos.capes.gov.br/10.1145/3132847.3133060>.
- [21] H. Zhu, X. Li, P. Zhang, G. Li, J. He, H. Li, and K. Gai, “Learning tree-based deep model for recommender systems”, in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '18, London, United Kingdom: Association for Computing Machinery, 2018, pp. 1079–1088, ISBN: 9781450355520. DOI: 10.1145/3219819.3219826. [Online]. Available: <https://doi-org.ez48.periodicos.capes.gov.br/10.1145/3219819.3219826>.
- [22] A. A. Kardan and M. Ebrahimi, “A novel approach to hybrid recommendation systems based on association rules mining for content recommendation in asynchronous discussion groups”, *Information Sciences*, vol. 219, pp. 93–110, 2013.
- [23] H. Song, H. Zhang, and Z. Xing, “Research on personalized recommendation system based on association rules”, in *Journal of Physics: Conference Series*, IOP Publishing, vol. 1961, 2021, p. 012027.
- [24] R. Halverson, “An empirical investigation comparing if-then rules and decision tables for programming rule-based expert systems”, in *[1993] Proceedings of the Twenty-sixth Hawaii International Conference on System Sciences*, vol. iii, 1993, 316–323 vol.3. DOI: 10.1109/HICSS.1993.284327.
- [25] H. Ishibuchi, T. Sotani, and T. Murata, “Tradeoff between the performance of fuzzy rule-based classification systems and the number of fuzzy if-then rules”, in *18th International Conference of the North American Fuzzy Information Processing Society - NAFIPS (Cat. No.99TH8397)*, 1999, pp. 125–129. DOI: 10.1109/NAFIPS.1999.781667.

- [26] H. Ishibuchi, T. Nakashima, and T. Murata, “A fuzzy classifier system that generates fuzzy if-then rules for pattern classification problems”, in *Proceedings of 1995 IEEE International Conference on Evolutionary Computation*, vol. 2, 1995, 759–764 vol.2. DOI: 10.1109/ICEC.1995.487481.
- [27] W. Li, W. Meng, C. Su, and L. F. Kwok, “Towards false alarm reduction using fuzzy if-then rules for medical cyber physical systems”, *IEEE Access*, vol. 6, pp. 6530–6539, 2018. DOI: 10.1109/ACCESS.2018.2794685.
- [28] F. Belqasmi, R. Glitho, and C. Fu, “Restful web services for service provisioning in next-generation networks: A survey”, *IEEE Communications Magazine*, vol. 49, no. 12, pp. 66–73, 2011.
- [29] H. Chong and F. L. Gaol, “Ula lab: Ubiquitous open contents web-based language laboratory using rest protocol web service”, *International Journal of Multimedia and Ubiquitous Engineering*, vol. 8, no. 4, pp. 199–206, 2013.
- [30] M. Viggiano, R. Terra, H. Rocha, M. T. Valente, and E. Figueiredo, “Microservices in practice: A survey study”, *arXiv preprint arXiv:1808.04836*, 2018.
- [31] D. Rubilar and A. Aguilera, “Automated menu recommendation system focused on clinical nutrition”, in *2019 IEEE CHILEAN Conference on Electrical, Electronics Engineering, Information and Communication Technologies (CHILECON)*, 2019, pp. 1–7. DOI: 10.1109/CHILECON47746.2019.8988061.
- [32] A. G. F. Mazuelos, N. R. Y. Pelaez, and E. A. Cerna, “Technological solution for the development and validation of a healthy diet in times of pandemic”, in *2021 IEEE 1st International Conference on Advanced Learning Technologies on Education & Research (ICALTER)*, 2021, pp. 1–4. DOI: 10.1109/ICALTER54105.2021.9675087.
- [33] *Nestjs*, Last accessed 02 November 2022, 2022. [Online]. Available: <https://nestjs.com/>.
- [34] *What is service-oriented architecture (soa)?*, Last accessed 07 April 2023, 2023. [Online]. Available: <https://www.ibm.com/topics/soa>.

- [35] *Python*, Last accessed 11 November 2022, 2022. [Online]. Available: <https://www.python.org/>.
- [36] *Pipenv: Python dev workflow for humans*, Last accessed 11 November 2022, 2022. [Online]. Available: <https://pipenv.pypa.io/en/latest/>.
- [37] *Postgresql*, Last accessed 15 November 2022, 2022. [Online]. Available: <https://stackshare.io/postgresql>.
- [38] *Postgresql: The world's most advanced open source relational database*, Last accessed 15 November 2022, 2022. [Online]. Available: <https://www.postgresql.org/>.
- [39] *Sqlalchemy*, Last accessed 15 November 2022, 2022. [Online]. Available: <https://www.sqlalchemy.org/>.
- [40] M. Makai, *Sqlalchemy*, Last accessed 15 November 2022, 2022. [Online]. Available: <https://www.fullstackpython.com/sqlalchemy.html>.
- [41] *Welcome to alembic's documentation!*, Last accessed 16 November 2022, 2022. [Online]. Available: <https://alembic.sqlalchemy.org/en/latest/>.
- [42] *Fastapi*, Last accessed 16 November 2022, 2022. [Online]. Available: <https://fastapi.tiangolo.com/>.
- [43] *Django vs flask vs fastapi – a comparative guide to python web frameworks*, Last accessed 16 November 2022, 2021. [Online]. Available: <https://analyticsindiamag.com/django-vs-flask-vs-fastapi-a-comparative-guide-to-python-web-frameworks/>.
- [44] *Django vs flask vs fastapi for software founders*, Last accessed 16 November 2022, 2022. [Online]. Available: https://dev.to/kateryna_pakhomova/django-vs-flask-vs-fastapi-for-software-founders-45k4.
- [45] P. Bansal and A. Ouda, “Study on integration of fastapi and machine learning for continuous authentication of behavioral biometrics”, 2022. DOI: 10.1109/ISNCC55209.2022.9851790.

- [46] *Pydantic*, Last accessed 16 November 2022, 2022. [Online]. Available: <https://pydantic-docs.helpmanual.io/>.
- [47] *Aquavitae app*, Last accessed 16 November 2022, 2022. [Online]. Available: <https://github.com/hmarcuzzo/aquavitae-app/tree/main>.
- [48] *Stack overflow*, Last accessed 16 November 2022, 2022. [Online]. Available: <https://stackoverflow.com/>.
- [49] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed. USA: Prentice Hall PTR, 2008, ISBN: 0132350882.
- [50] L. Barbosa and A. Hora, “How and why developers migrate python tests”, in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 538–548. DOI: 10.1109/SANER53432.2022.00071.
- [51] *Why pytest for writing functional api tests*, Last accessed 28 December 2022, 2014. [Online]. Available: <https://skimlinks.com/blog/why-pytest-for-writing-functional-api-tests/#:~:text=Why%5C%20use%5C%20Pytest%5C%3F%5C&text=Pytest%5C%20fixtures%5C%20makes%5C%20it%5C%20really,module%5C%20or%5C%20method%5C%20we%5C%20add..>
- [52] *Chatgpt*, Last accessed 04 April 2023, 2023. [Online]. Available: <https://openai.com/blog/chatgpt>.