Development of a database migration and compatibility system

Marcos Vinícius Alves Pereira

Dissertation presented to the School of Technology and Management of Bragança to obtain the Master Degree in Industrial Engineering.

Work oriented by:

Prof. Dr. Paulo Alexandre Vara Alves

Prof. Dr. Vasco Augusto Pilão Cadavez

Prof. Dr. Mateus Antunes Oliveira Leite

Bragança

2023

Development of a database migration and compatibility system

Marcos Vinícius Alves Pereira

Dissertation presented to the School of Technology and Management of Bragança to obtain the Master Degree in Industrial Engineering.

Work oriented by:

Prof. Dr. Paulo Alexandre Vara Alves

Prof. Dr. Vasco Augusto Pilão Cadavez

Prof. Dr. Mateus Antunes Oliveira Leite

Bragança

2023

Dedication

I dedicate this thesis to my grandparents who have always encouraged me to seek knowledge.

Acknowledgement

This research is funded by the European Food Safety Authority (EFSA), Grant Agreement GP/EFSA/BIOHAW/2022/01. The positions and opinions presented in this thesis are those of the author alone and do not represent the views or scientific works of EFSA.

Abstract

Biological hazards, such as bacteria, viruses, and parasites, pose significant threats to human health when encountered through inhalation, ingestion, or skin contact. The Pathogens-in-Foods (PIF) database was established to address these concerns, serving as a centralized resource for accessing information about food borne pathogens in Europe.

The European Food Safety Authority (EFSA) plays a pivotal role in ensuring food and feed safety within the European Union. However, the EFSA and PIF databases employ different data structures, making compatibility and data migration essential to enhance the relevance of PIF.

This thesis focuses on the development of a data migration and compatibility system, allowing seamless communication between the PIF and EFSA databases. Through this system, the databases can align their information more effectively, ultimately improving food safety and public health.

Using Python data migration scripts, it was possible to migrate the old PIF database to EFSA standards, making the two databases compatible. The script was developed following a system of classes, making it possible to use these same classes for future migrations if necessary or even migrations to other databases.

The compatibility system developed offers the possibility of converting variables between different databases without the need for data migration. This system is especially useful for variables or entities that cannot or should not undergo the data migration process. In this way, the system operated as a conversion system, making PIF entities compatible with EFSA variables.

Keywords: Biological Hazards; Data Migration; Food Safety; Database Integration

Contents

Acknowledgement **vii**

Abstract **ix**

1 Introduction **1**

1.1 Context 2

1.2 Objectives 2

1.3 Typographic Norms 3

2 State of art and Study of tools **5**

2.1 Ontologies 6

2.1.1 Food Ontologies 7

2.1.2 Pathogen ontologies 7

2.2 Database types 8

2.3 Data migration 9

2.3.1 Data migration with non-relational databases 10

2.4 Compatibility system 11

3 The PIF and EFSA systems **13**

3.1 PIF database 13

3.1.1 Study 15

3.1.2 Study structure 16

3.1.3 Bacteria 17

3.1.4 Bacteria structure . . . 20
3.1.5 Parasite and virus . . . 24
3.1.6 Parasite and virus structure . . . 26
3.1.7 Countries structure . . . 28
3.1.8 Food structure . . . 29
3.1.9 User . . . 31
3.2 PIF Front-end . . . 32
3.2.1 Search page . . . 33
3.2.2 Search by label page . . . 36
3.2.3 Register New page . . . 38
3.2.4 Curate Data page . . . 41
3.3 PIF back-end . . . 42
3.4 The European Food Safety Authority (EFSA) . . . 43

4 System methodology **46**
4.1 System methodology overview . . . 46
4.2 Compatibility issue related to different variable names . . . 47
4.3 Compatibility issue related to different hierarchies . . . 48
4.4 Compatibility issue related to the codes . . . 50
4.5 Proposed solution . . . 51
4.5.1 Document structure diagrams . . . 51
4.5.2 Data migration . . . 52
4.5.3 Compatibility system . . . 53

5 Development **55**
5.1 Document structure diagrams . . . 56
5.2 Data migration . . . 58
5.2.1 Migration script . . . 58
5.2.2 Side scripts . . . 64
5.2.3 Manual migration . . . 66

5.3 Compatibility system 68
5.3.1 Compatibility system side implementation 72

6 Results **75**
6.1 Understanding of structures 75
6.2 Formulation of the migration script 76
6.3 Compatibility System 77
6.4 Addressed Issues 78

7 Conclusion and Future Work **81**

A Protocol for the Pathogens in Foods **87**

B Migrations script **89**

List of Tables

4.1 PIF and EFSA variable names 48
4.2 ESFA Codes 51

List of Figures

3.1 PIF Database squema 15
3.2 Study record 16
3.3 Study structure 17
3.4 Bacteria record 18
3.5 Bacteria record with multiple methods 19
3.6 The main bacteria structure 21
3.7 The structure of the Count object 22
3.8 The structure of the Prevalence object 23
3.9 Bacteria types structure 24
3.10 Common fields structure 25
3.11 Parasite record 26
3.12 Virus record 26
3.13 The main parasite structure 27
3.14 The main virus structure 28
3.15 Countries collection 29
3.16 Food structure main object 30
3.17 Food structure common object 31
3.18 User record 32
3.19 Search page drop-downs 33
3.20 Search page second set of drop-downs 34
3.21 Search page results table 34
3.22 Search object 35

3.23 Back-end response . . . 36
3.24 Search by label page . . . 37
3.25 Search by label object . . . 37
3.26 First section of the *Register New Page* . . . 38
3.27 Dynamic rendering 1 . . . 39
3.28 Dynamic rendering 2 . . . 39
3.29 PIF object groups . . . 39
3.30 Cascade drop-downs snap code . . . 40
3.31 Curate Data page . . . 41
3.32 Curate Data table . . . 41
3.33 EFSA Catalogue browser *Hierachies* mode . . . 44
3.34 EFSA Codes . . . 44
3.35 EFSA Catalogue browser *Facets* mode . . . 45

4.1 Methodology diagram . . . 47
4.2 PIF *PackStatus* structure . . . 49
4.3 EFSA *PackStatus* structure . . . 50

5.1 EFSA legume structure graphical representation . . . 57
5.2 Graphical representations for new structure of Legumes . . . 58
5.3 New proposed structure . . . 65
5.4 Migration rules . . . 65
5.5 New dairy structure . . . 67
5.6 Old dairy structure . . . 68
5.7 PIF navigation bar . . . 69
5.8 Search EFSA page . . . 69
5.9 Search EFSA page query . . . 70
5.10 Compatibility object array . . . 70
5.11 Compatibility object . . . 73
5.12 Compatibility box . . . 74

Chapter 1

Introduction

Biological hazards can be understood as agents (bacteria, viruses, parasites) that can threaten human health when inhaled, eaten, or in contact with the skin. Within this context, thinking especially about food, the Pathogens-in-Foods (PIF) was developed. These days, there are countless sources available for accessing information. When it comes to information related to pathogenic agents, it is no different. On one hand, it is easier than ever to access information. However, on the other hand, this abundance of information can often be highly disorganized and lacking in harmony. The PIF was created to address this problem, creating a common place, to consume data related to pathogens in food in Europe, in an orderly and user-friendly way.

In a broader range, the European Food Safety Authority (EFSA) is the agency in the European Union responsible for covering all matters related to food and feed safety. To fulfill its purpose EFSA has it's pattern and metrics regarding storing information about food and pathogens in food. As a separate creation, the PIF database has also it's own patterns, which do not necessarily meet exactly with the one used by EFSA. In this context, the development of a data migration and compatibility system that makes the two databases communicate equally is fundamental to make PIF even more relevant.

1.1 Context

The proposal of this thesis aims to solve the compatibility issue between PIF and EFSA databases. The concept is using technologies and ways of converting the PIF database into EFSA requirements or even developing a compatibility system that will operate making the PIF data EFSA compatible.

1.2 Objectives

1. Development of a database migration and compatibility system: Create a system that encompasses all the stages of migration and compatibility between two databases. The system will operate as a method that includes the graphical visualization of the databases, the actual migration of the data, and finally, a system for reconciling variables and entities that will not be migrated.

2. Creating graphical representations of the databases: the use of graphical representations of the entities present in both databases and how they relate to other entities plays a significant role in the process of understanding how the databases work.

3. Migration of old records from the PIF database to EFSA standards: the process of migrating all the old records in the PIF database to EFSA terminologies.

4. Migration of PIF structures to EFSA standards: the part responsible for changing the structures that write the PIF data to EFSA standards, ensuring that the new data entered into the PIF will follow the EFSA standard.

5. Back-end and front-end adjustments to adapt to the new structures: given the change of structures in the PIF database, possible adjustments to the application's back-end or front-end will be necessary.

6. Creation of a compatibility system: system responsible for making variables and entities compatible between PIF and EFSA without data migration, used in cases

where data migration cannot or should not be carried out.

It's important to understand that the development schedule of the thesis presented above has a shortened period of execution, running from January to September instead of the usual September to September. This occurred due to a change of topics during the thesis's execution. A research grant was opened within the theme developed in this thesis, causing the old topic to be abandoned and the new one to be adopted.

1.3 Typographic Norms

In order to facilitate better text readability while also referencing variables, collection names, methods, and related elements, a simple convention was adopted during the writing of the thesis. Any word with its meaning linked to some kind of external variable had its font written in italics. For example: *bacteria*, *parasite*, and *virus* when referring to collections with the same names. Another commonly used example throughout the thesis was the use of italics to reference variable names within figures.

Chapter 2

State of art and Study of tools

The process of data migration can be simply understood, as the name implies, as the process of moving data from one place to another or even modifying the structure of an old database into a new structure. In order to perform this process many variables need to be analyzed, such as the present database structure and technologies which is the goal to achieve with this migration.

In this sense, a solution can be thought of in three steps: ontologies; data migration, and a compatibility system. Each one of these steps aims to take a certain part of the whole data migration process. Splitting the complexity of the migration process into steps can not only ease the process but also help each one of those parts to be carried out more thoroughly.

Thinking about each step of the process independently, the ontologies are the part responsible for providing a better understanding of the current database and how the database should look after the process of migration. As the author [1] affirms, ontologies are used to represent data as a set of concepts, the ontologies are also based on a graphical representation, which helps immensely in understanding a database structure.

The database migration step is where the migration process happens, but for that two main points need to be taken into consideration: database types and database migration technologies. In this sense, this steep talks about understanding the database that will

go through the process of data migration, so to say, this database is a relation or a non-relational one? Which type of database technologies are used? On the other aspect is also important to realize which technologies will be used to carry out the process of data migration, this point is dependent on the technologies used in the database and the best technologies available today.

Lately, the compatibility system is responsible for carrying out what could not or should not be done by the migration the data migration process. Given the complexity of some database structures and the requirements for the database migration, based mainly on the time required to carry out this process, some steps of the process can be chosen to not be migrated, or simply they can be migrated given the system. For this reason, the compatibility system aims to fix data-related problems without carrying out a data migration, solely with the use of mapping of variables.

2.1 Ontologies

First, to better understand what ontologies are, is necessary to understand the objective in creating them. According to [2], ontologies aim at capturing knowledge about a specific domain and providing common accepted representation that may be re-used and shared by diverse applications and groups. In the words of [3] ontologies are also a means to document the structure of a particular domain, which helps to develop a common understanding of its concepts. In agreement with [1] ontologies represent knowledge as a formal description of a domain of interest.In this sense, by understanding the goal behind the creation of an ontology is possible to understand its use and applications.

An ontology allows to describe databases information given a semantic dimension to the entity relations.This means that it is possible to enrich the information of a database and also infer new knowledge. Viewing a database as a distinct domain, where knowledge is encapsulated in a structured data format, following a specified schema, is a fundamental step in aligning a database with the broader context of ontology descriptions. By considering these concepts, we can effectively represent all the elements within a database

as ontological entities and derive valuable insights from this approach [2] [1] [3].

2.1.1 Food Ontologies

The work in [4] discusses the use of ontologies to achieve the so-called FAIR principle (findable, accessible, interoperable and reusable) within agrifood. In [4] the importance of ontologies to represent a domain of knowledge and how this can help in creating standardized definitions for terms is discussed. Another important point made by [4] is that there is no general ontology covering all aspects of agrifood, which contributes to the creation of isolated ontologies and the fragmentation and isolation of data. In this context, [4] argues for the use of quality ontologies and the possible use of collaborations to address complexities in the ontology creation process.

In the work of [5] an effort is made to develop an ontology about fast foods. Understanding the impact that information about a given food can have on individuals, [5] seeks to develop a standardization proposal for the presentation of fast food data. Using a sample of 21 establishments, an ontology is created using the OWL2 language and software Protégé, both of which are widely known in the field of ontologies. With the studies in [5] it is possible to understand the informative role of ontologies and to understand the step-by-step process behind their development.

In [6] is discussed about the FoodOn a food ontology that aims to standardize information related to foodborne pathogens. The authors of [6] highlight the challenge in sharing information regarding foodborne pathogens given the large number of different platforms, data dictionaries, and free-text descriptions. Ultimately the work of [6] presents the FoodOn as a solution to standardized data related to food and shows the importance of such a feat in improving data sharing.

2.1.2 Pathogen ontologies

The work of [7] deals with the creation of a generalized system for the storage and retrieval of phenotypc information for bacteria. In order to standardize the capture of information

on microbes, the ontology created by [7] also tries to remain compatible with other existing ontologies within the same scope. During the work on [7], the terms present in the well-known GO (Gene Ontology) were used in order to create a new ontology that could be used as the standard for representing phenotypc information. The authors of [7] used the ontology editor OBO-Edit to create their own ontology.

In [8] are proposed terms to describe plant diseases caused by fungi and oomycetes that do not count in the Gene Ontology (GO). The efforts of the authors of [8] go around identifying behavior and process related to the plant disease that has no correspondence in the Gene Ontology, from that they developed 256 new terms and extant other 38 from the the GO. An important aspect in the work of [8] is the effort to understand what could not be presented by the GO terms and then the proposal of new terms that would better suit the scenarios they were aiming to display.

As stated by [9] nowadays microbiological surveillance is supported by public health organizations, with systems that integrate multiple metadata related to food-borne disease and various other sectors. The work of [9] focuses especially on the "One Health Structure in Europe" (COHESIVE) that aims to integrate pathogen information from various sections such as public health, animal health, and food safety. In this sense, the authors rely on the FoodOn ontology to describe information related to food while developing their web-based platform called CIS (COHESIVE Information System). Work [9] presents a relevant attempt to unify information related to pathogens in different sectors.

2.2 Database types

Before understanding the process of data migration is important to first grasp the types of databases that exist and the main difference between them. The database type (relational or non-relational) plays a big role in the data migration process, especially when a migration occurs from a non-relational to a relational database or vice-versa. Within the range of data migration with the same type of databases, understanding the used type is also a very relevant factor.

Relational databases based on the concepts of sets in mathematics, all the data represented as mathematical n-ary relations, an n-ary relation being a subset of the Cartesian product of N domains, in the words of [10]; still in the definition of [10], this data model is very specific and well organized. Columns are described by the well-defined schema. The set of related data stored in rows has the same structure.

Regarding non-relational databases [10] provides the following description: the common and the main feature that distinguishes the NoSQL data model is it does not use the table as a storage structure of the data. In this context is possible to understand that relational databases use a rigid structure regarding the store of their data, such as tables while non-relational databases use a more free structure, not relying on tables for storing their data.

2.3 Data migration

Data migration is the selection, preparation, extraction, transformation, and permanent movement of appropriate data that are of the right quality, to the right place, at the right time, and the decommissioning of legacy data stores, to deliver the business transformation aspirations of the organization, per the findings of [11]. Through this definition is possible to understand what is data migration and its role, it is also possible to go further and understand each one of the four points enumerated.

The selection is relative to finding exactly what will be migrated, or which part of a whole; preparation, extraction, and transformation can be comprised in a single step, responsible for dealing with the data that will go through the process of migration, working on this data to make it ready. Regarding the two last steps: permanent and movement, they line up together in a way that one should ensure that the change is permanent, that is, there's no way back and the other should ensure the movement of change itself [11].

2.3.1 Data migration with non-relational databases

Narrowing the scope of the database, thinking only about non-relational ones, in the work of [12] presents a framework for data migration only with NoSql (non-relational) databases. In [12] first is shown the type of NoSQL existing, namely: key-value; document; column, and graph. Each one of these types stores data in a non-relational way but in different manners. Given those differences, [12] work in a general framework to approach the data migration within NoSQL databases.

Based on the study by [13], is proposed a model for data migration between NoSQL databases. Focus on cloud-based NoSQL data stores, in the work of [13] a proposal is made approaching the problem regarding heterogeneity among different NoSQL databases. Through the development of a series of algorithms, the process of data migration from a document-based database into a graph one could be completely automatized. With this, is possible to have a glimpse of the problems involving data migration within the NoSQL domain and how they can be approached.

According to the research by [14], in the context of NoSQL databases, a self-adapting data migration is proposed. The idea is that in times of agile software development, the NoSQL databases often need to deal with multiple versions of stored data, which needs to be handled by the code of the application in most cases. In [14] is proposed an approach to automatlcy deal with migrations, through the use of self-adapting methodology. An important fact described by [14], is related to the 4 types of data migration strategies, namely: the eager migration strategy; the lazy migration strategy; the incremental migration strategy; and the predictive migration strategy. Each one of the strategies described is shown to fit better in a given context.

The eager migration aims to migrate all legacy entities at once, so anytime the data model is changed, all the related entities are also changed. This process results in two main outcomes: a consistent database at any time and a higher migration cost. Even though the benefits associated with the eager migration are big, so are their costs. The laze migrations operate with the opposite strategy as the eager migration, so in this case,

legacy data will remain unchanged even when there are changes in the data model. The changes using the lazy migration approach will only happen on the fly when an entity is accessed that requests the new data model. This approach has a smaller cost but a higher structural entropy that tends to increase over time [14].

The incremental migration can be understood as the middle ground between the eager and lazy migration. The incremental migration treats the data model changes as a lazy migration, only changing an entity when this is accessed, but also performing periodic clean ups in the database migrating the legacy entities to fit the new data model. The predictive migration aims to go one step further than the incremental strategy, it operates with periodic clean ups, but with a fundamental difference, the separation of data into "hot" and "could". This strategy seeks the entities that are more used for an application, the so-called "hot" data, and proceeds with the migration for these; the "cold" data, or entities not so frequently used are not migrated immediately but on-the-fly, or when is necessary. This approach proved the perfect balance between migration costs and latency (how quickly data can be accessed) [14].

2.4 Compatibility system

Understanding a compatibility system as nothing more than mapping the variables to other variables. Studies on the semantic web offer an interesting reference point for mapping variables. In the specific case of the semantic web, mapping occurs with the intention of promoting the conversion of a given database into an ontology model.

The work in [15] proposes a model for converting data from a relational database into RDF (Resource Description Framework). Framework). The proposal works with two mapping layers, the mapping layer and the template layer, dividing the mapping into two parts. The first part is responsible for the mapping between SQL and RDF; the second layer is responsible for the design and how this mapping will be presented. With regard to the mapping layer proposed by [15] , it is possible to use the same reasoning on different fronts.

According to the research by [16] a tool is proposed for automatically mapping a database into an ontology. Also starting from a relational database, the tool created works by first understanding the types of tables present in the database in question and then creating a mapping process. The mapping process begins by analyzing the type of table, and grouping it into three possible categories. The grouping takes into account the number of foreign keys in the table and their dependencies. With the tables properly grouped, the mapping process takes place by synchronizing database entities with classes and sub-classes in the ontologies. In this way, it is understood that the mapping proposal in [16] works by first grouping entities to subsequently create generic solutions that fit into each of the grouped categories.

The study of [17] uses the two main methods for mapping databases to ontologies, direct mapping and wrapping methods.The former is based on creating a new ontology from the database, while the latter is based on using legacy ontologies, mapping the database to an existing ontology. The methods described are compared and the positive and negative points of each are noted. Through the research of [17], with the use of didactic images, it is also possible to gain a better understanding of how the entities of a database are mapped onto ontologies.

Building on the studies of [18] [19], the pattern described as Lookup Tables operates like a conditional if-else or a switch case but uses an object instead of a code block. In this case, it is possible to create a large number of conditionals to map one variable to another without data migration while still maintaining good response times.

Chapter 3

The PIF and EFSA systems

3.1 PIF database

Firstly to understand the problem approached by this thesis, it's necessary to understand why the problem emerged foremost. The Pathogens-in-Foods (PIF) system, as described in the introduction of this thesis, works with data collected from papers to gather and store this data was necessary to create a pattern. A researcher will analyze the study thoroughly before categorizing it as suitable for being inserted in the PIF database; a certain level of relevance and methodological quality will be mandatory, then the data will pass through a process of systematic categorization of microbiological methods, food types, and outcomes.

In the conception of the PIF database, the setting for constraints related to variable names, food hierarchies, agents hierarchies, and database structure was based only on the experience of the researchers in charge of the project. The researchers conducted the project independently; for this reason, they had a great degree of freedom regarding making changes in the patterns established by themselves.

In the course of the project, as the researchers continued reading more articles and, therefore, inserting more data in the PIF database, they noticed the need to make changes and adjustments to fit their goal with the project. The project continued to undergo minor

changes until the agreement with EFSA, where from that point on, the EFSA pattern for data storage was to be established upon the previous PIF pattern.

The PIF application is composed of three parts: the database, the front-end, and the back-end. Among these three, the database plays the most relevant role within the application, understanding its behavior it's crucial to understanding the overall behavior of the application. To have a better grasp of the database it is important to analyze every collection that compose it, how it's structured, and the role this collection plays.

The collections within the PIF database can be categorized into two types: structural and non-structural. The structural collections are responsible for recording the information in the database and the non-structural for retaining the data. Thus, the database works with collections pairs, and this pattern applies to nearly all collections, except for a few cases that will be subsequently explained.

Figure 3.1 shows the schema of the PIF database. You can see that the relationship between *study* and the agents *Bacteria*, *Parasite* and *Virus* is one-to-many, while there is no constrictive relationship between the *User* collection and the other collections. So, each of the collections contains its corresponding structural part responsible for writing the information to the database and a part responsible for being the record in the database. As such, the scheme shown in Figure 3.1 looks at the aforementioned collections in terms of themselves as pairs made up of a structural part and a record part.

The *Food structure* and *Countries* collections, which will be explained below, behave differently from the other collections mentioned, where there is no attempt to use the concept of primary key and foreign key. The collections have a dynamic role, where both are used as the rendering of one or more inputs on the front-end that will represent possible variables within the agent's collections. As such, the *Food structure* and *Countries* collections don't exist on their own, playing a purely drop-down rendering role

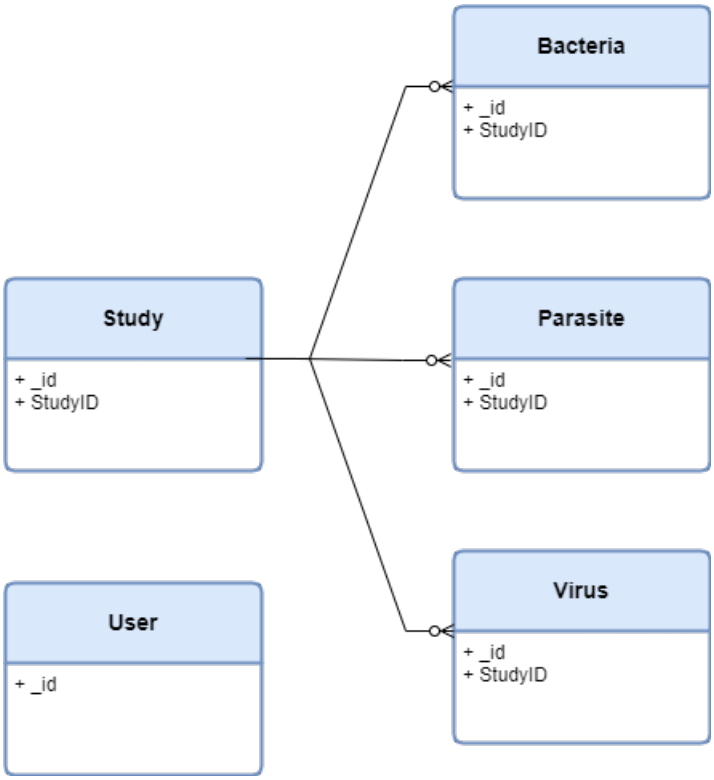

Figure 3.1: PIF Database squema

3.1.1 Study

The *study* collection is fundamental for the PIF application despite the fact of only representing one single input in the front-end. As was shown in the previous collections, all of the nonstructural collections related to the agents had a value for *StudyID*, which is the variable responsible for linking the „study“ collections and the collections related to the agents (*bacteria*, *parasite*, *virus*).

Within the PIF application for one given study is possible to exist many records, but one record can only be linked to one study, creating a relation one to many. Following the expected flux to insert a new record, a study needs to be inserted first, so that later on this study can be linked to the newly inserted record for a given agent. A separate page is used in the front-end only to insert new studies, while the already inserted studies, the ones present in the *study* collection, are displayed trough one single drop-down menu.

Figure 3.2 shows a record inside the *study* collection.

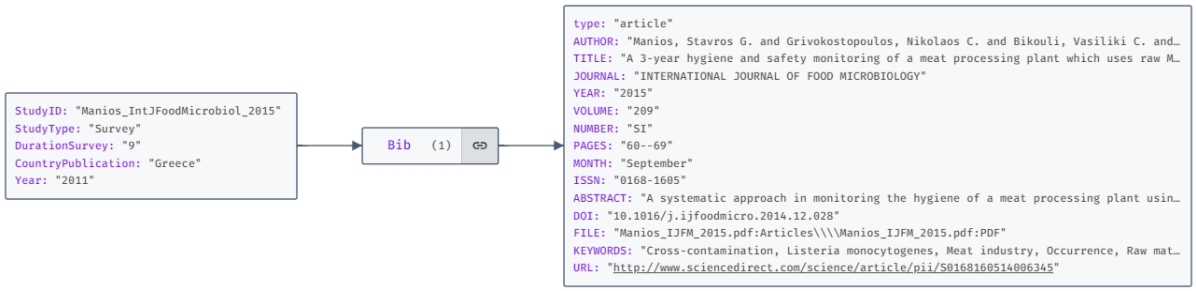

Figure 3.2: Study record

3.1.2 Study structure

The behavior behind every one of the structural collections in the PIF database is very similar, some collections may have more or less impact on the overall application and its complexity. The structural collections for the studies follow the design of the other structural collections already shown. In this type of design, an object is displayed in the front-end as numerous input boxes or drop-downs based on its fields. All the configurations necessary for the front end to know how to deal with the given input are stored in the database.

Figure 3.3 shows how the structure of the studies is assembled. In this example is quite evident the use of the *data* variable to tell the front-end if the referred field should be displayed as an input box or a drop-down.

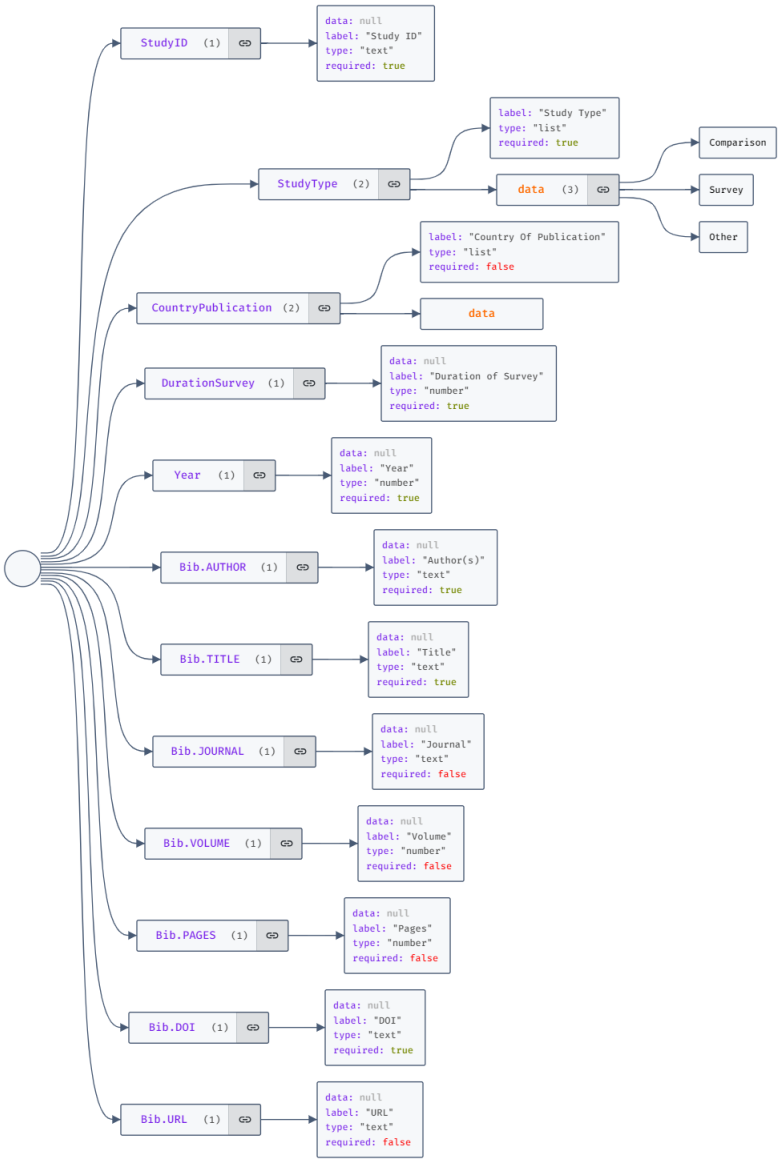

Figure 3.3: Study structure

3.1.3 Bacteria

The PIF application works with three kinds of pathogens: *bacteria*; *virus*; and *parasite*. Each one of these pathogens has its own set of collection pairs (structural and non-structural) that interact with other collections within the database. From these interactions emerge the relationships between the collections. These relationships offer a

glimpse into how the system works.

The *bacteria* collection is the one that holds the largest number of records inside the database, Its behavior, in terms of database, is very similar to the other pathogens. As usual in a non-relational database, the records within every collection are represented as JSON objects, to facilitate understanding, these objects were displayed in this thesis using a graphical representation. Figure 3.4 presents what an individual record inside the bacteria collections looks like.

For each record in the bacteria collection, there are mandatory and non-mandatory fields. The number of fields in an individual record can vary within a certain range. The process of writing the mandatory and non-mandatory fields, as well as the rules for such, will become clear when analyzing the *bacteria-struct* collection, the corresponding structural pair of the present collection.

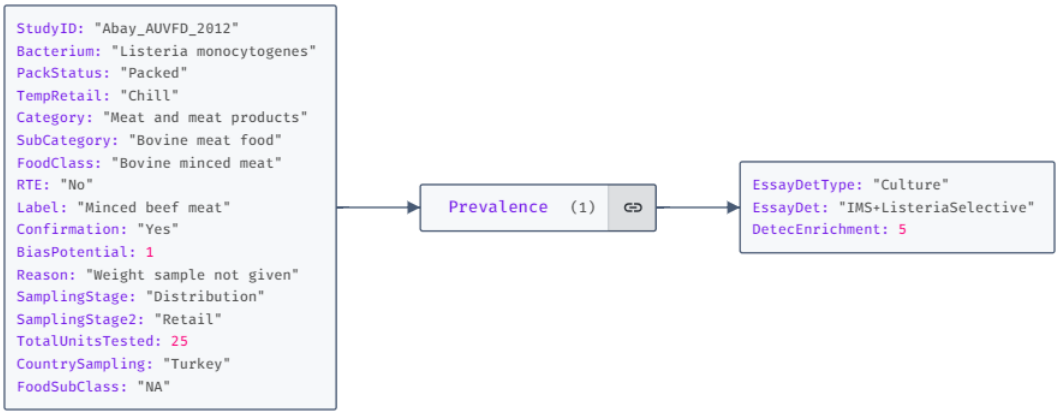

Figure 3.4: Bacteria record

By investigating Figure 3.4 it is possible to verify how a bacteria record is composed. The field *StudyID* is the one responsible for creating the link between the *bacteria* collection and the *study* collection. They have a One-to-Many type of relationship, where one study can be present in multiple bacteria records but the bacteria record is always linked to only one study.

Fields such as *Bacterium*; *PackStatus*; *Category*; and *SamplingStage* are always mandatory, but they contain sub-levels that vary based on the chosen option. For some options the user will have more or less mandatory sub-levels to choose. This will become evident when exploring the food structure collection and understanding how this system is displayed in the front-end.

The variable that holds the value for *Prevalence* has a special behavior: it stores the information in the database as a nested object. This nested object per se has its field and, as will be evident while analyzing the structural collection, its rules. Referring to 3.5, it is possible to see that there's also another nested object called *Count*.

The fields *Count* and *Prevalence* are extremely relevant for the bacteria records, they're also an example of how the mandatory field can vary largely upon what is selected. Based on the current implementation is mandatory for a bacteria record to have at least a *Count* or a *Prevalence* value.

On a broader scale, it is evident that the *bacteria* collection only plays the role of storing the data. The complexity of the PIF application lies in other collections that play an active role in how the data are displayed in the front-end and how the user can interact with that upon the constraints established by these collections.

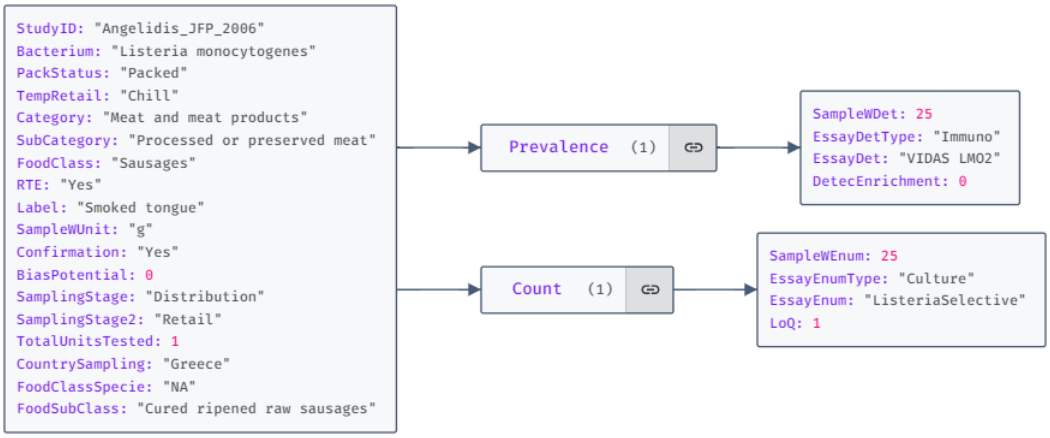

Figure 3.5: Bacteria record with multiple methods

3.1.4 Bacteria structure

The front-end display rules have a strong connection with the structural collections in the PIF database. The structural collections are displayed in the front-end as drop-down menus or input boxes, where each field of individual objects inside the collections represents one individual drop-down or input box. The objects inside the structural collections contain all the options that can be written by the system, each field of the objects represents an option that can be chosen.

Given the large number of possible options to choose from and the expected behavior for a given agent (bacteria, virus, parasite), the front-end is responsible for consuming these structural collections and displaying them in an ordered way, following certain rules for displaying that will become evident once analyzing the front-end behavior.

In the PIF application, there are separate pages for each one of the agents, so the page responsible for bacteria will only consume the collections related to bacteria and the common collections. Entering pages related to bacteria, the applications always have a page responsible for writing the data, a page responsible for reading the data and displaying it (search page), and a page responsible for editing inserted data. The structural collections are primarily utilized on the writing and editing page; nevertheless, they also have a minor role on the search page.

Figure 3.6 shows the main object inside the bacteria structural collection. This object is responsible for storing the main used fields related to bacteria, as well as the *Count* and *Prevalence* objects with their respective fields. The objects for *Count* and *Prevalence* are extremely complex and their behavior needs to be volatile, so to speak. For that reason, the fields related to the *Count* and *Prevalence* objects are scattered among various objects inside the bacteria structural collection.

Illustrated in Figure 3.7 is the main object for the *Count* method. The actual object has a substantially greater number of fields; however, for pragmatic elucidation, only a select few fields have been used in this depiction. The fields of this object have the same behavior as the previously mentioned main structural bacteria object, but here all the

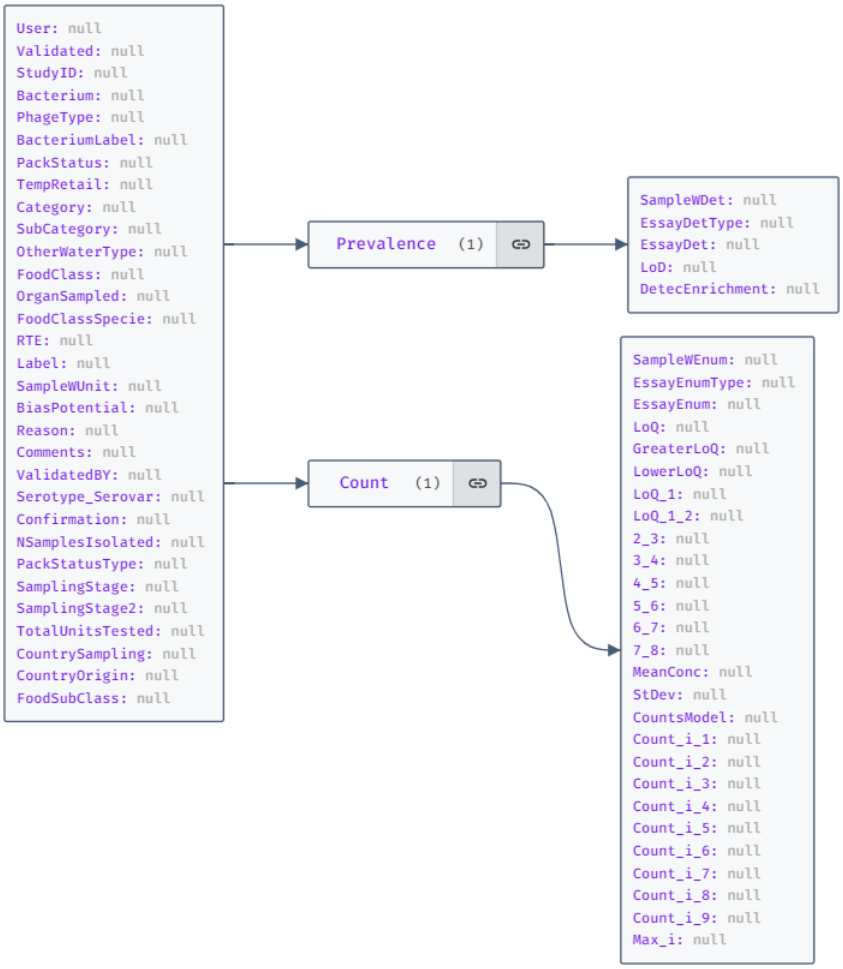

Figure 3.6: The main bacteria structure

fields of the main object are also objects, creating a nested object structure.

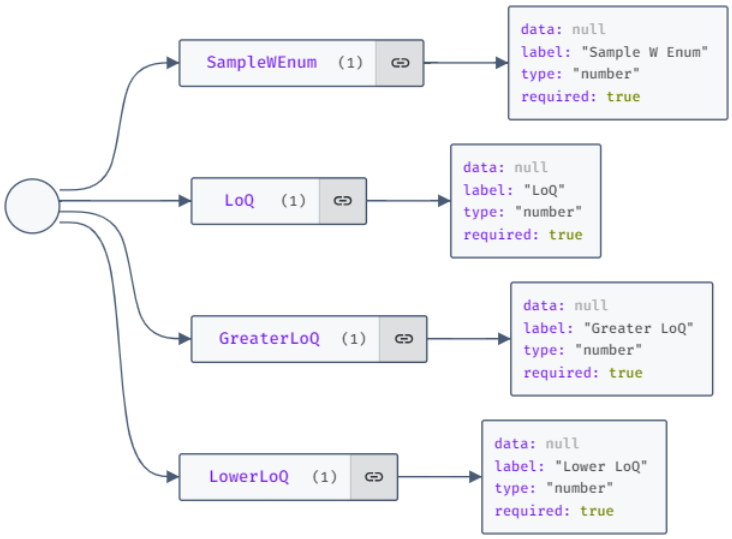

Figure 3.7: The structure of the Count object

Depicted in Figure 3.8 is the main object for the *Prevalence* method. In this case was possible to depict the object in its full fields once the object was considerably smaller than the *Count* one. One interesting fact to notice about the objects shown in Figure 3.7 and Figure 3.8 is how the structure inside each one of the nested objects works.

The structure consists basically of four fields: *data*, label, type, and required. The *data* field will indicate to the front-end if the field is to be rendered as an input box or as a drop-down. Setting the value of the *data* to null indicates that the field will be rendered as an input box. The *label*, *type*, and *required* fields indicates precisely what their names imply.

Another very important structure in the present collection is the type of bacteria object. For simplification reasons, Figure 3.9 only shows one of the many bacteria types available inside the full tree. To provide a given behavior for the *Count* method and another type of behavior for the *Prevalence* method each one of the types of bacteria has its pair of methods.

For the system to handle the multiple options existing when approaching different

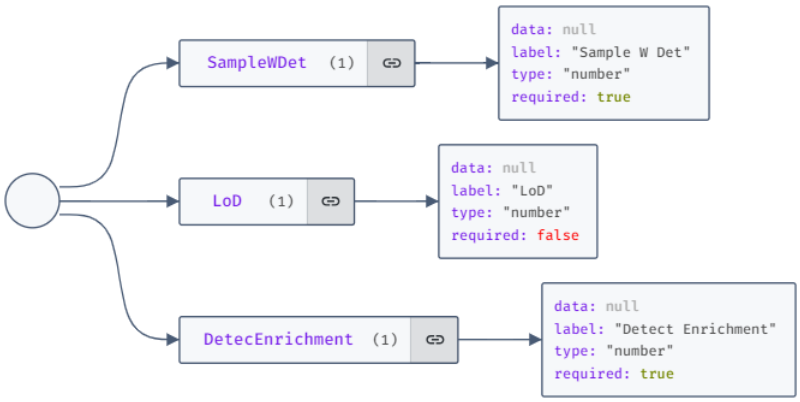

Figure 3.8: The structure of the Prevalence object

agents, it was necessary to implement a conditional rendering in the front-end, a lot of this rendering relies on the way those objects are structured in the database. The logic applied for this object will be mirrored for *virus* and *parasite*. In biological terms, these agents can be immensely different, but in database terms, they have almost the same behavior with more or less fields.

The object depicted in Figure 3.10 has the same behavior as the other objects mentioned, it's possible to see that in this case, the *data* field is not null, making this field being rendered in the front-end as a drop-down menu. The data field used in this case was an array, but in other parts of the application, the *data* field was populated with an object. The use of arrays in this scenario made the structure more stiff. In a structure based on a nested object it's easier to increase or decrease the number of sub-levels or how nested the structures are, with arrays the level of freedom is hardly achieved.

The PIF database was built following a decentralized pattern in almost every sense. The structure in Figure 3.10 is populated with common fields for all agents, to follow a decentralized pattern, this same structure was built in the collections responsible for the virus and parasite structure.

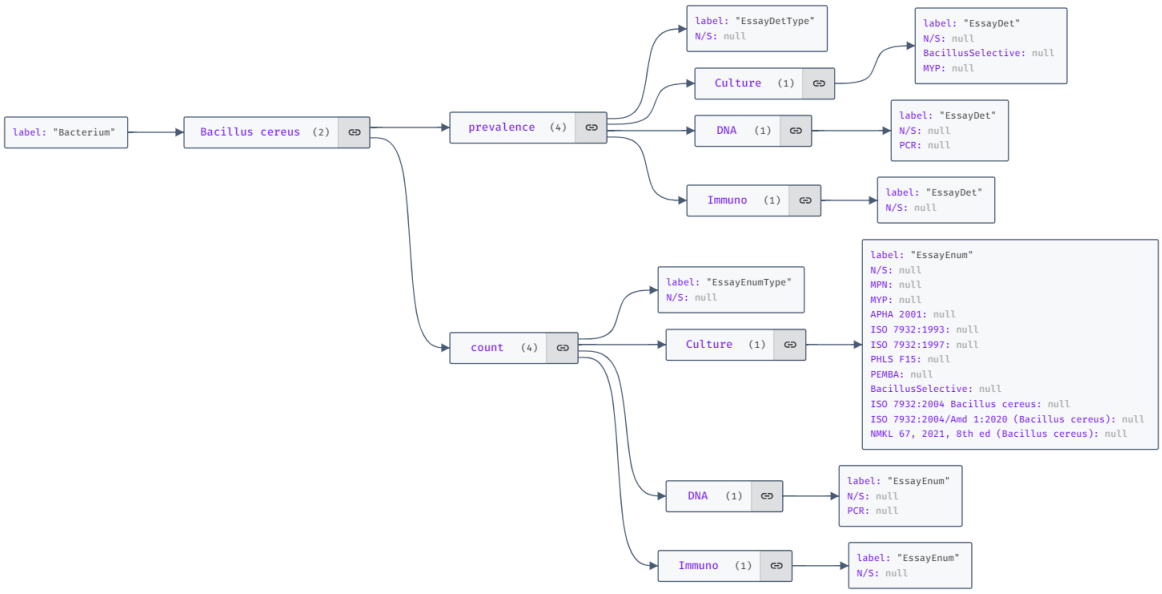

Figure 3.9: Bacteria types structure

3.1.5 Parasite and virus

The collections *parasite* and *virus* work in the same way as the collection *bacteria*. The purpose of these collections in the database is only to store the data, so this data can be consumed and displayed on the search page in the PIF application. Overall the collections related to parasites and viruses have a greater number of fields and objects, due to the complexity of those subjects.

Displayed in Figure 3.11 is possible to see what a record in the *parasite* collections looks like. The records do not differ in any way from the *bacteria* records except for the greater number of fields.

Figure 3.12 displays an example of a record in the *virus* collections. The behavior for the pages responsible for the *virus* and the *parasite* collections in the front-end are almost the same, showing more complexity than the page responsible for the *bacteria* collection. The greater numbers of the field in these mentioned collections imply that their structural pair will have, unconditionally, more field, which results in a great number of inputs to be generated in the front-end. The number of inputs in the front-end already increases

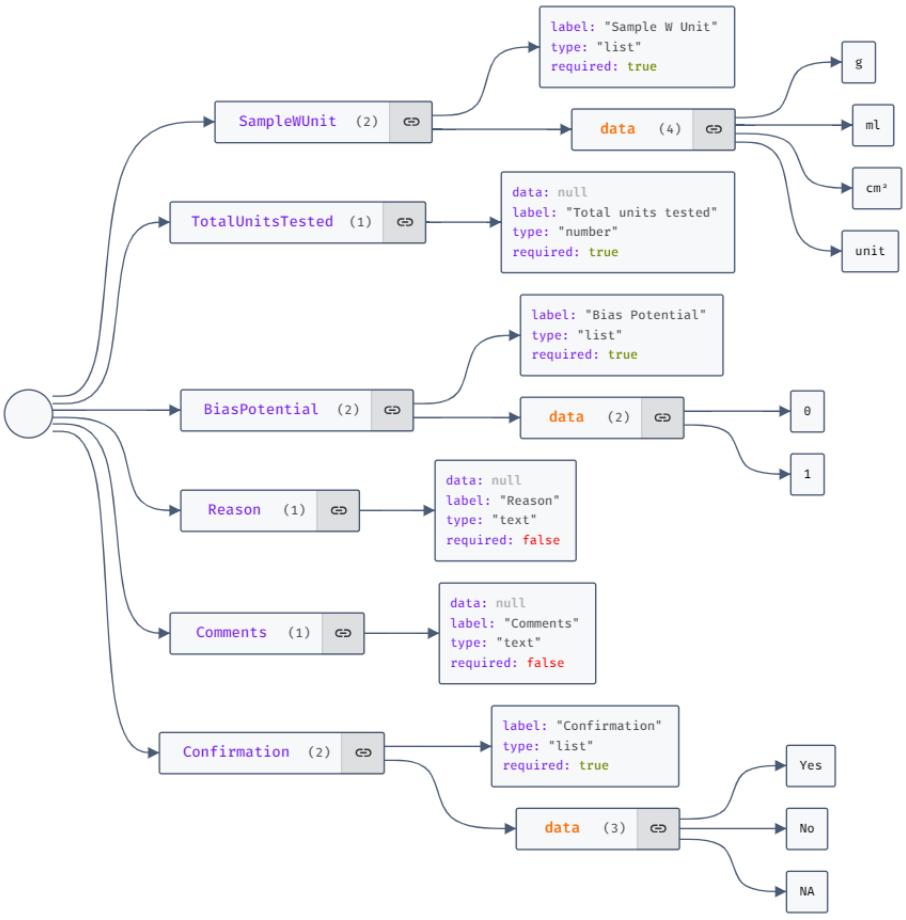

Figure 3.10: Common fields structure

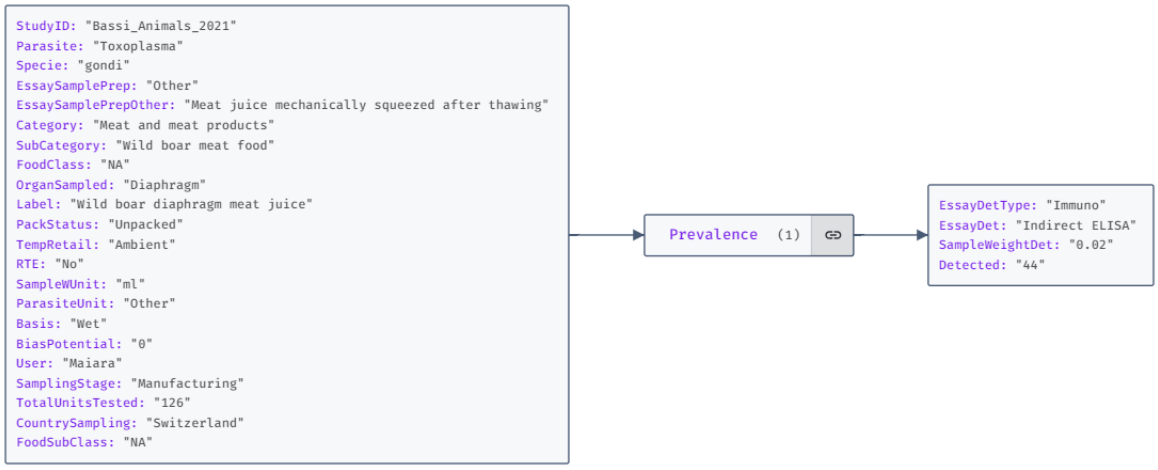

Figure 3.11: Parasite record

the complexity of the page, the conditional rendering for some of those inputs makes the complexity even higher.

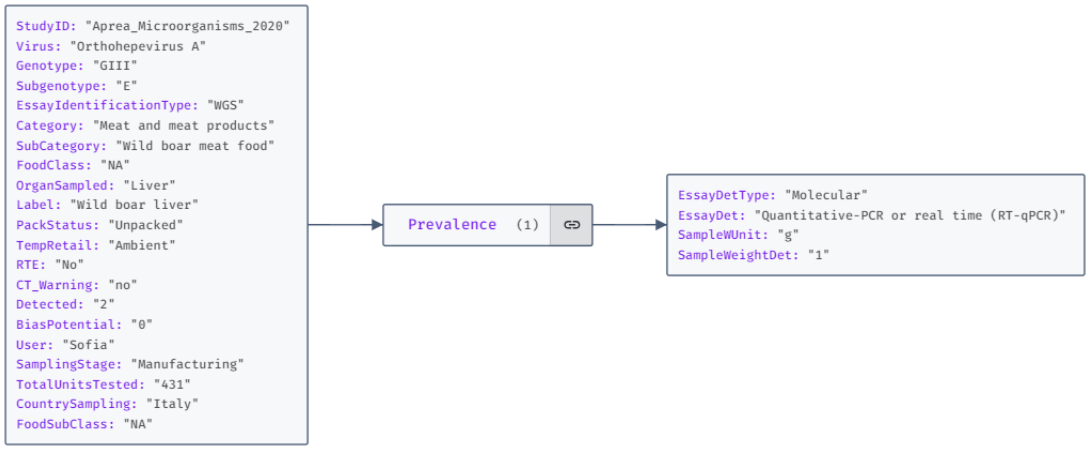

Figure 3.12: Virus record

3.1.6 Parasite and virus structure

Both main structures responsible for the structural part of the *parasite* and *virus* collections are very similar. The rest of the structures present in those two structural collections

are very similar to what was already shown in the section that covers the bacteria structural collection. As the database follows a decentralized model, the structural collections for bacteria, parasites, and virus have a giver number of repeated structures.

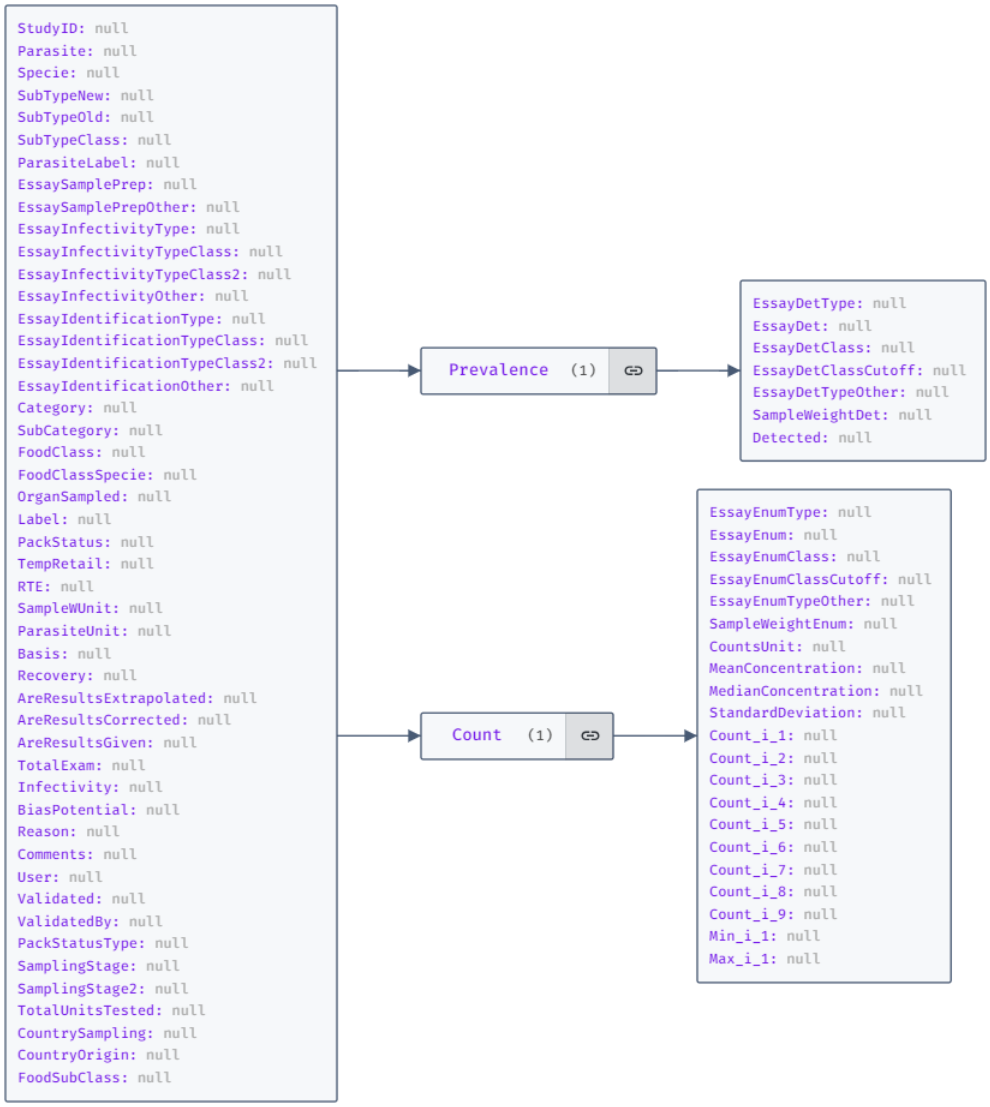

Figure 3.13: The main parasite structure

Figure 3.13 and Figure 3.14 show how the main object for the parasite and virus structural collection looks like. By analyzing the picture is possible to notice how the *Count* and *Prevalence* methods for these collections are more complex when compared

to the ones for bacteria. This greater number of fields results in more complexity in the front-end of the application.

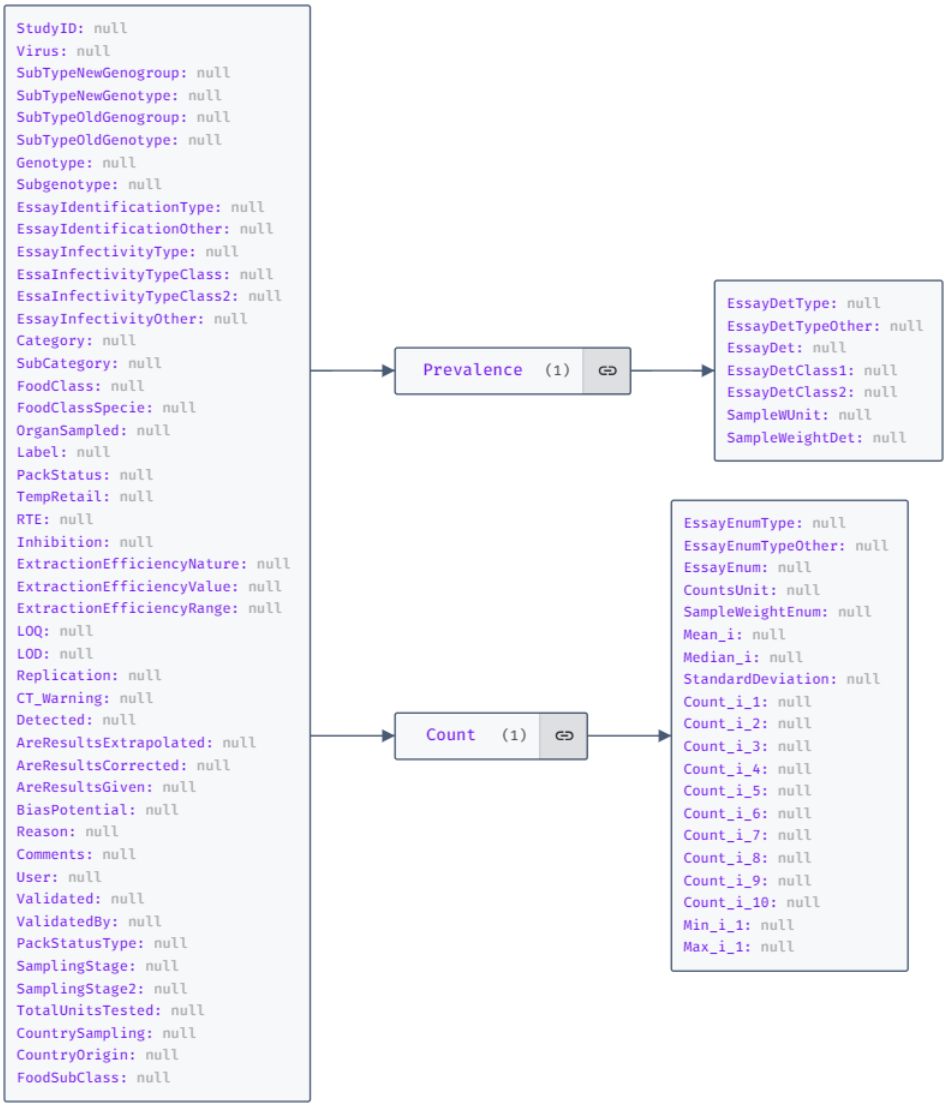

Figure 3.14: The main virus structure

3.1.7 Countries structure

The behavior of this collection is very straightforward and does not play a big role in terms of increasing the complexity of the system. Each one of the inserted records in the

PIF database must have a value for the country of origin of the given study, therefore, this collection provides a list of countries for attributing this value. The entire collection is rendered in the front-end as one single drop-down menu, this drop-down can be reused in many parts of the application to display the same functionality but always consume the same collection.

Looking at Figure 3.15 is possible to see what the record in the *countries* collections looks like. In terms of the interaction of the user, while inserting or consuming data from the PIF applications, the only variable that represents meaning is *Country*. The other variables are only used in a feature using Shiny that also consumes the PIF Database to show in a map the places from where the inserted studies come.

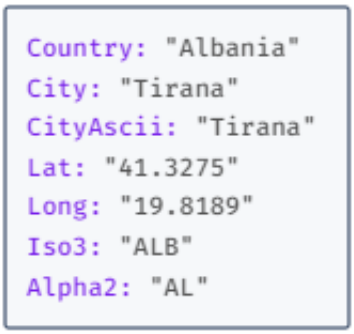

Figure 3.15: Countries collection

3.1.8 Food structure

The collection related to the food structure is the only collection in the database that only has their structural part, not working in pairs as most of the collections already showed. As the intention behind the build of the PIF applications was to store data related to pathogens in foods, there is no relations need for the database to have a non-structural food collection. All the records added to the PIF database will have mandatory some relation with the food structure collection.

The food structure collection is the most complex of all the structural collections shown. As depicted in Figure 3.16, it is possible to see how the structure behaves as a nested object, this structure is especially important to provide the correct display of this object in the front-end. The image illustrated in Figure 3.16 shows only a part of the whole food structure that counts with many other fields.

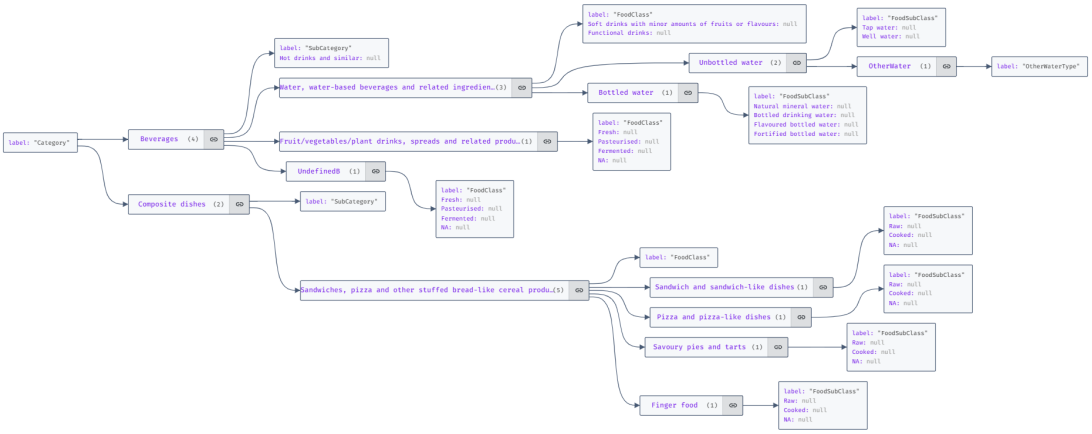

Figure 3.16: Food structure main object

Analyzing Figure 3.16 makes it possible to understand how is the behavior of the food structure in the front-end. The nested object inside the food structure follows four levels: *Category*, *SubCategory*, *FoodClass*, and *FoodSubClass*; these four levels follow a hierarchy going from the *Category* to the *FoodSubClass*. This hierarchy created in the database makes it possible to render a cascading drop-down in the front-end, making a series of conditional renderings. This solution allows a reduced number of code lines in the front-end, eliminating the usage of conditional and allowing a more lean code.

The object depicted in Figure 3.17 shows the second and last object present in the food structure collection. This object follows the same patterns already shown for the other structural objects. The front-end uses this object to render food-related properties. As for the object depicted in Figure 3.16 or Figure 3.17, they are once rendered by the fronted and later the variables corresponding to their field are saved in the agent's non-structural

collections.

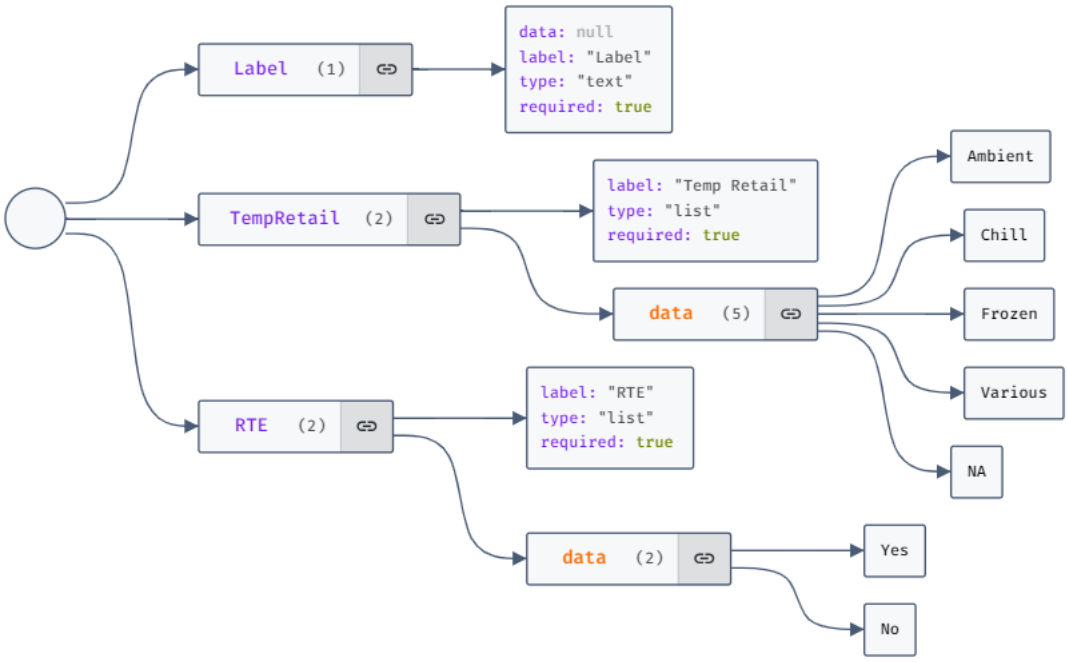

Figure 3.17: Food structure common object

Subsection Title

3.1.9 User

The *user* collections work in a slightly different way than the other pairs of collections already mentioned. For this collection, there is no corresponding structural one, once this responsibility is performed by the back-end directly. The records saved in the database have a direct relation with the user collections, once there is an option implemented in the front-end that allows an admin user to available data inserted by a non-admin one.

Figure 3.18 shows a record inside the *user* collection. The most relevant variable of any record inside this collection is the *Role*, through this variable the system knows which features will be unblocked and which won't. Also using the variable *Role*, an authorized user can evaluate data inserted by a basic user, by doing that, the user *GivenNames* will

Figure 3.18: User record

be linked to the given record.

3.2 PIF Front-end

Even though most of the work carried out by this master's thesis was in the database of the applications, often changes in the database would reverberate in a non-expected behavior in the front-end. Given this fact, to fully understand the problem approach of this thesis and how it was attacked is also important to have a brief overview of the front-end and its functions.

The front-end of the PIF application works with the notorious JavaScript library *React.js*, configured to work also with *JavaScript*. Given the complexity of a *React.js* application, this section aims to explain what in this part of the application is relevant to the work of this thesis.

Overall we can understand the front-end of the PIF application within four different pages (*Search*, *Search By Label*, *Register New*, and *Curate Data*) that will be repeated to all agents (*bacteria*, *parasite*, *virus*) and also for the studies. The decentralized approach in the database reverberates in the front-end behavior. By understating the behavior of these pages is possible to grasp how the front-end interacts with the database and its importance to the operation of the application.

3.2.1 Search page

The *Search* page is one of the most important ones in the PIF application, through it the user can access the data in the PIF database. The idea behind the PIF project was to have a place where the data, related to pathogens in foods, could be accessed. Therefore, the *Search* page plays a relevant role, within the goals of the application, by displaying the data.

The behavior of the *Search* page can be understood as the normal behavior of any given search page, the user interacts through some sort of filter or input to have back the stored data. In the PIF application, the user can choose from some drop-down menus which data he wants to retrieve, and apply multiple filters to refine his search.

Figure 3.19 shows the first set of drop-downs where the user can filter the data he wants to retrieve from the database. The operation of the *Search* page is completely done with the closed field, so the options to be chosen are limited by the number of options inside of each drop-down. While executing a search operation the user is free to select a single option for each one of the desired drop-downs, every one of those options will sum together as a *and* conditional.

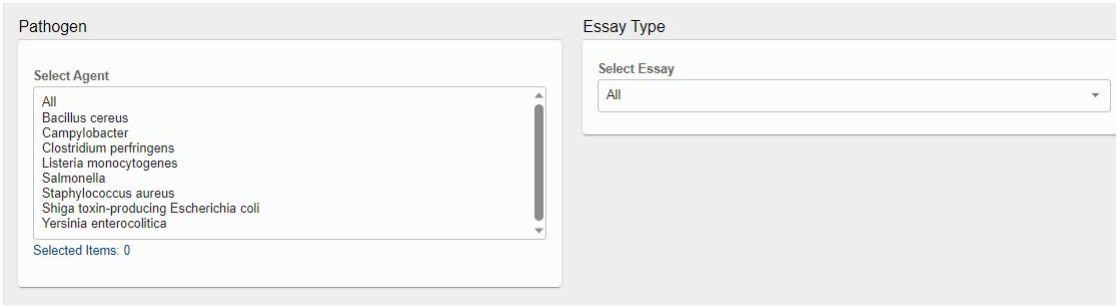

Figure 3.19: Search page drop-downs

3.20 depicts the second set of drop-downs available for the user. In the *Search* page there is also a third set of drop-downs when the user marks the option *Show Advanced Filters* illustrated in Figure 3.20. Another relevant fact to notice in Figure 3.20 is the options available for the data output, the user can choose to display the data as a table or make the download as a *CSV* file or as a *JSON* File. It is also possible to select how

the empty cells should be treated, filled with blank fields, undefined or null.

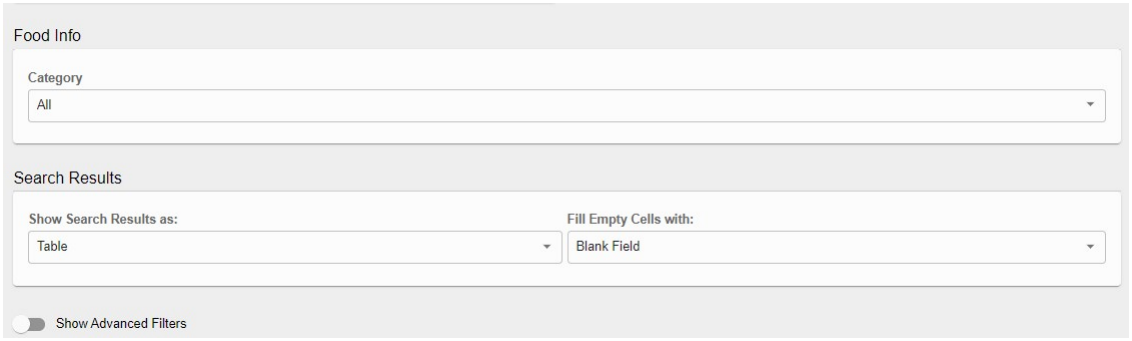

Figure 3.20: Search page second set of drop-downs

Figure 3.21 shows how a search operation looks like in the PIF applications, in this case, the search results are displayed as a table with a blank field for the empty cells. The structural objects for each one of the agents (bacteria, parasite, virus) are consumed in the search page to provide the drop-downs and its respective values.

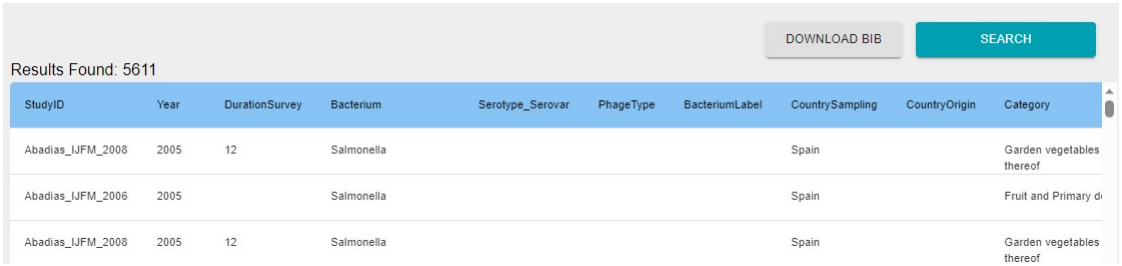

Figure 3.21: Search page results table

Understanding the behavior of a search page is fundamentally understanding the request traffic. The front-end solicits the back-end for data, both from the front-end to the back-end, and from the back-end to the front-end. The data must transit in a way that allows the two fronts to communicate understandably. Comprehending how the front-end and back-end of an application communicate is key to fully grasping the application's operation.

Figure 3.22 exhibits the object built by the front-end to request data from the back-end. The object is composed of many fields that will have a value based on the options the user selects on the search page. When an option of one of the search page drop-downs is

selected, this value will populate a field of the request object, otherwise, the field related to this specific drop-down will remain empty.

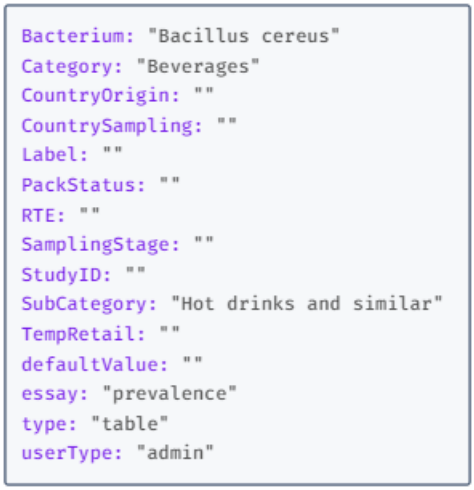

Figure 3.22: Search object

The information returned by the back-end has a different structure than the information sent to the back-end, Figure 3.23 illustrates the returned data is an array structure. The return data model works by sending an array of arrays where the first array is the header and the rest of the arrays the body. This information is used by the front-end to populate the header of the displayed table and then the corresponding rows. The figure depicted in 3.23 is just a cut of a normal response that can extend to over forty fields per array, and have endless nested arrays.

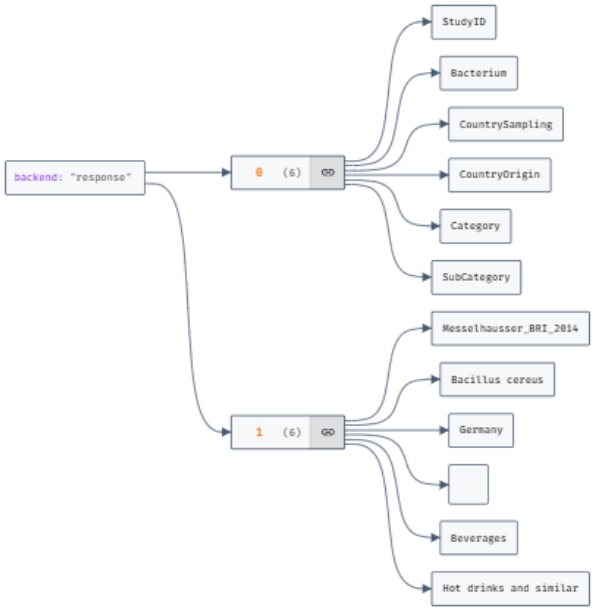

Figure 3.23: Back-end response

3.2.2 Search by label page

The operation of the search by label page is very similar to the search page, with the advantage of allowing the search in an open field in the PIF application called *Label*. The *Label* field is mandatory when a new record is inserted in the database. Different from most other fields, its value is not given through a drop-down menu but through an input box. In the scenario, the user is free to insert any type of information related to the data

being inserted.

Figure 3.24 shows the appearance of the search by label page. The user is still allowed to filter the requested data using some of the drop-down menus but can also use the open field to search in the database. Figure 3.25 shows an example of a search where the word *melon* is attributed to the variable *Label*. In this case, all the records in the database that contain the word *melon* in their *Label* value will be returned. It is important to notice that the function implemented in this page executes a partial text search, also called, fuzzy search, where the search is executed in records that contain a specific word.

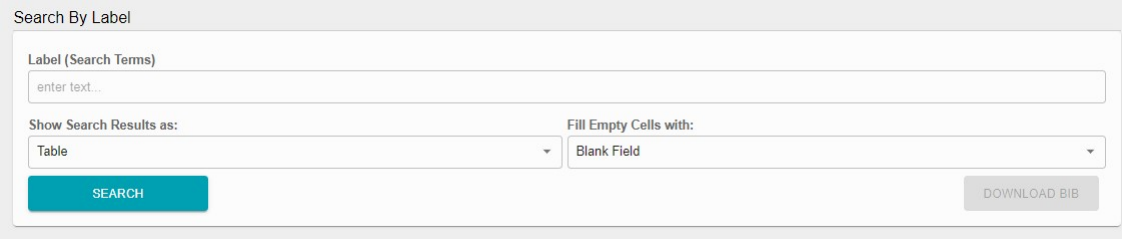

Figure 3.24: Search by label page

Figure 3.25: Search by label object

3.2.3 Register New page

The *Register New Page* is the most complex one in the PIF application. To be able to understand its behavior is necessary to grasp how the page is structured and also how the React code operates underneath the front-end surface. All the pages responsible for inserting data in the application have the same behavior varying only in the degree of complexity of its components and functionalities. The figures used in the section show the page use of the *bacteria* collection; however, this same model will be mirrored for all the other pages related to data inputting.

The basic functionality of the insert pages in the PIF application is given through the use of drop-down menus. Each one of the drop-downs used in the PIF application is rendered from an object stored in the database. Looking at Figure 3.26 is possible to see how the objects inside the bacteria structural collection are rendered in the front-end as drop-down menus. Figure 3.26 also shows how the inserted record is firstly linked, necessarily, with a study by using the*Study ID* drop-down.

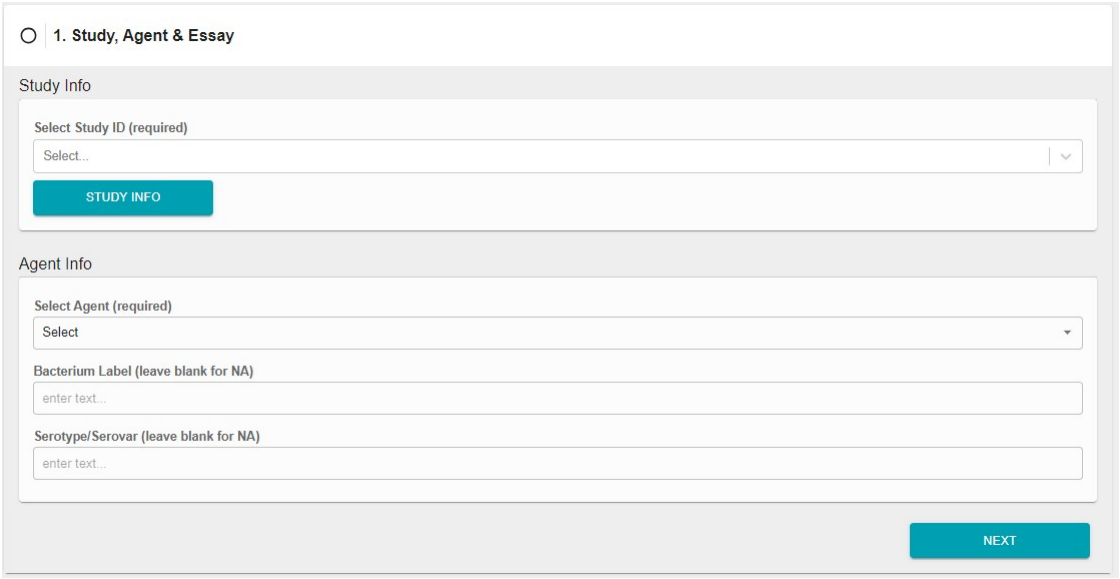

Figure 3.26: First section of the *Register New Page*

Figure 3.27, and Figure 3.28 show how conditional rendering is given in the front-end of the applications. Based on the option selected by the user different objects are

rendered in the front-end; those different objects have different nested objects inside, giving a great number of possible drop-downs to render. This characteristic gives the necessary dynamism to the PIF application to deal with different types of records based on the agent or food being inserted.

Figure 3.27: Dynamic rendering 1

Figure 3.28: Dynamic rendering 2

Figure 3.29 depicts in detail how the set of objects are grouped. While inserting data in the PIF application all the data related to a given record will be fitted into three groups: *Agent and Essay*, *Food*, and *Results*. These categories individually gather a given number of objects that have common characteristics between them.

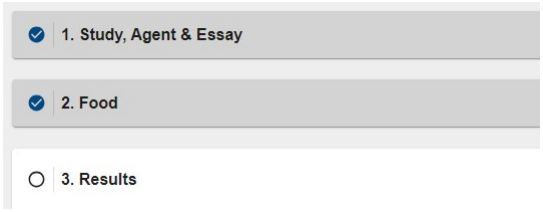

Figure 3.29: PIF object groups

```
(dataAuxEssayCount.map((json,index) => {
  return (
    <MaterialNativeSelect
      key={json.label}
      label={json.label + required_text}
      labelError={blank_text_error}
    >
      <option value={""}>Select</option >
      {Object.keys(json).map(key =>
        key != "label" && key != "selected" ? (
          <option key={key} value={key}>
            {key}
          </option >
        ) : null
      )}
    </MaterialNativeSelect>
)}))
```

Figure 3.30: Cascade drop-downs snap code

Figure 3.30 illustrates an important snap of code of the front-end responsible for the cascade loading of the drop-down menus. The pattern for rendering data showed in this snap code is used all over the PIF application, for both inserted and search-related pages. As was shown in previous sections, the drop-down objects are also rendered in the search pages but with a different purpose.

It's very important to notice the role played by the *dataAuxEssayCount* variable. This variable is the array responsible for storing all structural-related information, from this data the map function is used to iterate over each one of the objects inside of the array to render each one of them using the *MaterialNativeSelect* component. The second map function is used to iterate over the field of the given object to display them as options in

the drop-down menu. It is also possible to see how the values stored in the collections in the database for *Label* are now dynamically used in the front-end.

3.2.4 Curate Data page

The behavior of the *Curate Data* page can be understood as a mix between the search and the insert page in the PIF application. On the curate page, the user can search for a specific record in the database, select the given record, and then edit this record following the necessary adjustments. Figure 3.31 shows the input box used to search the data, Besides the input box, the user can also select which variable it wants to search based on the given input. In the shown example the variable *StudyID* was used to search.

After searching for specific data, the user can then select the row corresponding to the record and proceed to edit the record, Figure 3.32 illustrates how this process happens. When processing the edit page, the user will have a page that is the same as the one used for the insertion of data, with the only difference being that all the previous data related to this given record will be already populated in the drop-down menus.

Figure 3.31: Curate Data page

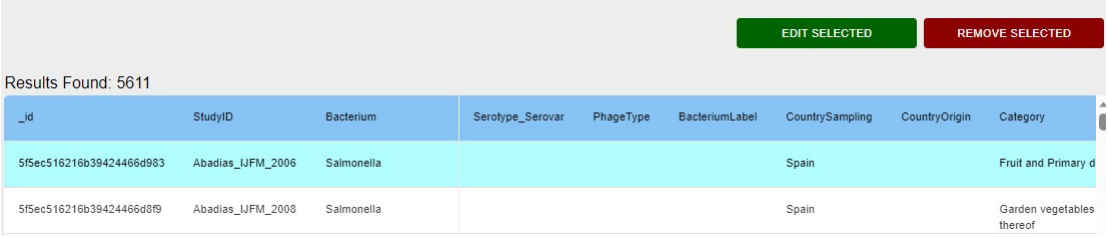

Figure 3.32: Curate Data table

3.3 PIF back-end

Within the scope of this present thesis, no changes were made in the back-end of the application. It's necessary nevertheless to have a quick overview of the principal technologies used in the back-end to enhance the understanding of the work done in the front-end and the database. The pattern used in the construction of the PIF application has its focus mainly on the interaction of the user with the front-end and then with the data storing given by this interaction. In this case, the back-end only plays the role of delivering and receiving the data between the two ends of the applications.

The back-end of the PIF application uses *Node.Js* to interact with the *MongoDB* database. The back-end, in turn, works with the framework *e Express* to simplify the creation of the APIs used in the application. Another worthwhile technology used in the back-end is the *Mongosee* library which helps in the interaction between the back-end and the database. In architectural terms, the back-end works with an MVC (Model-View-Controller) pattern, where the controller component is separated into four components: *Add*; *List*; *Manage* and *Struct*.

Each collection shown in previous sections has its own set of controller and controller modules, in a way that all of the controllers work in a very similar way. The *Add* module is the one responsible for processing HTTP POST requests in the main storing collection for the agents (*bacteria*, *parasite*, *virus*). The *List* module is responsible for the HTTP GET requests and for formatting the information that will be delivered to the front-end in a certain format: table; CSV or JSON.

The *Manage* component is responsible for handling the HTTP DELETE and POST (related to the data update) requests. Both of these methods are used on the curation page to enable the user to delete a record with necessary and to update a given record. The process of updating a record in the PIF application occurs by using a GET method to retrieve the previous information of the record; populating the drop-down menus in the front-end with the previous information, and using a POST method to write again the same record.

The *Struct* component assumes the role of rendering the HTTP GET requests related to the structural collections of the PIF database. As was shown in previous sections, the structural collections are displayed in the front-end as drop-down menus or input boxes; the *Struct* component in the back-end is the one responsible for getting this data from the database to enable the front-end to consume and render this data later.

3.4 The European Food Safety Authority (EFSA)

The European Food Safety Authority (EFSA) is an agency responsible for providing scientific advice and talking about food risks within the European Union (EU). The main focus of EFSA is on handling everything related to food a feed safety, things as animal health, plant protection, and nutrition. In this way, they help the EU to make safer decisions regarding food, even offering all their information openly to the public.

For the purpose of storing data related to food and feed safety, EFSA has its parameters and patterns. These parameters and patterns are kept in EFSA in catalog form, which is then made available by them through their web page for other users to access. Using the Catalog Browser the user can explore EFSAs that are directly connected to the Data Collection Framework where data gets submitted to EFSA [20].

Figure 3.33 illustrates what the EFSA Catalog Browser looks like. Through the use of this application is possible to search for a specific term, and see how this term is represented in the EFSA patterns. An important fact to notice is how the hierarchy of the database in the EFSA application can be displayed, the user can select between two options: *Hierarchies* or *Facets*; Figure 3.33 depicted the data displayed as *Facets* where Figure 3.35 depicted the data displayed as *Hierarchies*.

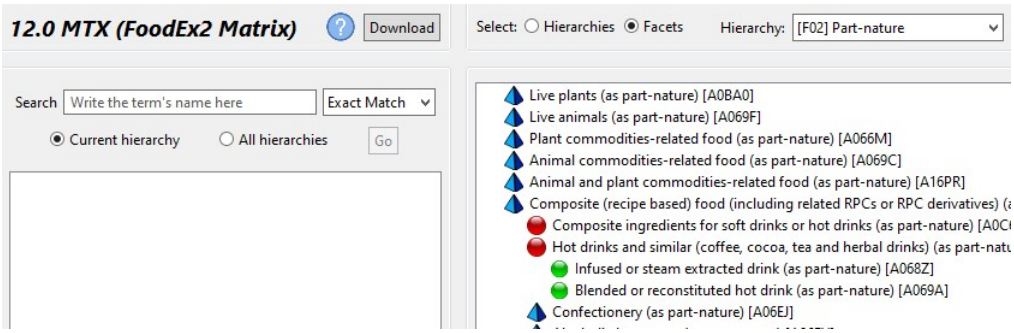

Figure 3.33: EFSA Catalogue browser *Hierachies* mode

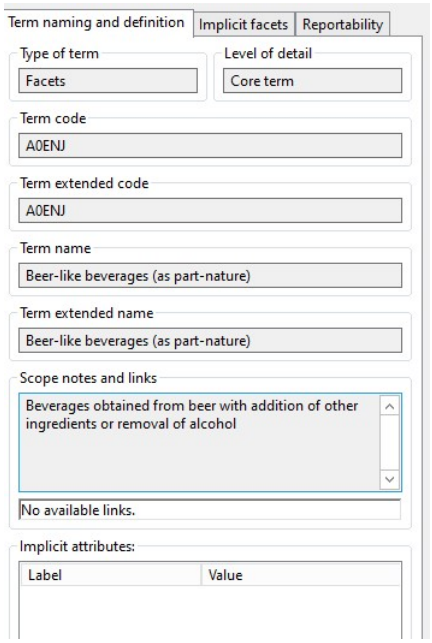

Figure 3.34: EFSA Codes

Regardless of the displayed mode selected, every term associated with the EFSA catalog has its code, as illustrated in Figure 3.34 in *Term code*. Understanding the pattern utilized by ESFA to store and display its data is fundamental to understanding the problem approach by this thesis and how it was approached.

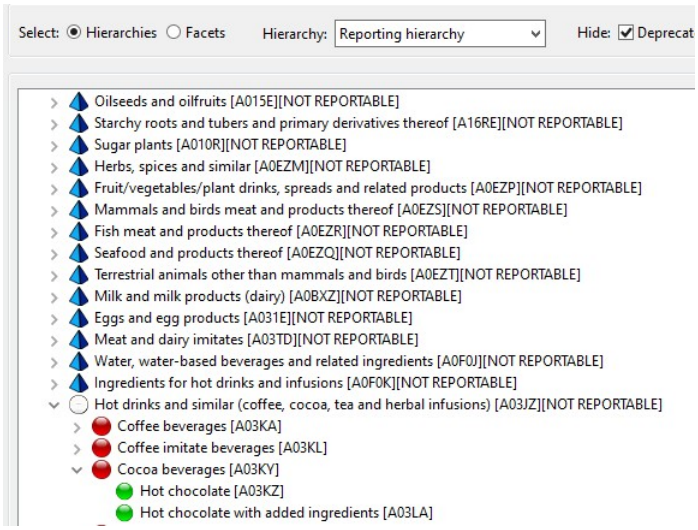

Figure 3.35: EFSA Catalogue browser *Facets* mode

Chapter 4

System methodology

4.1 System methodology overview

Looking at the diagram shown in Figure 4.2, it is possible to see how the methodology for this thesis came about. The first and most important step in the process is precisely the development of a database migration and compatibility system, where the whole idea behind the thesis is thought out, this part is related to the conceptualization of a system, an operating mode for the problem faced. Once the system has been thought out, you can then move on to the subsequent stages that will ensure that the system is implemented.

In the diagram in Figure 4.2, step 2 is part of the preparation, so to speak, for the subsequent processes; the graphical representations of the databases help to increase their level of understanding of them. Having understood the databases and the specifications of the desired changes, we then move on to stages 3 and 4, where the migration takes place. In this sense, the migration can be understood in two stages: the migration of the old data, the records from the old database that will be changed to a new desired standard (step 3), and the changes to the structures that write the data to the database (step 4). In this way, not only will the data in retrospect follow the new desired pattern, but also new data will already be written following the new pattern.

Step 5 represents the possible corrections that must be made to the front-end and

back-end of an application after a data migration. Sometimes the change in structures of the old bank can trigger malfunctions in the front-end or back-end of the application, which must then be corrected. In step 6, a compatibility system is created that makes entities in the two databases compatible without the need for data migration. This step is essential for data that should not be migrated or is chosen not to be migrated.

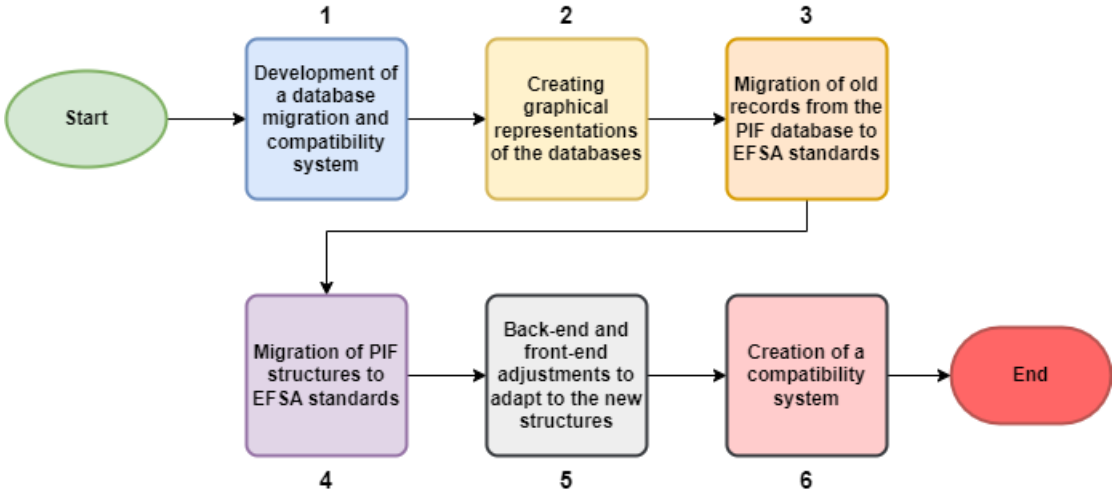

Figure 4.1: Methodology diagram

The entire thesis revolves around understanding the incompatibility problems between the PIF and EFSA databases and then proposing a solution. In this context, the compatibility problems can be thought of separately, dividing a larger problem into several smaller ones. Therefore, the following sections demonstrate the compatibility problem between PIF and EFSA on three different levels.

4.2 Compatibility issue related to different variable names

The first and more evident issue related to the integration between the EFSA and PIF databases was the difference in naming variables. In the development of the PIF database, the variable names were chosen, mainly, taking into consideration the experience of the

researchers in charge of the project; on the other hand, EFSA chose their variable names following different metrics. This event resulted in databases that resonate within the same subject, but follow different patterns.

Table 4.1, illustrates one example of the difference in naming variables between EFSA and PIF. When analyzing the table is possible to notice, how the EFSA went for the precise academic names for the terms related to bacteria and the PIF went for reduced and short versions. This difference in naming variables, even if that's not so striking, results in non-direct compatibility between the two databases. The problem illustrated by Table 4.1 was repeated many times for other variables within the EFSA and PIF databases.

Table 4.1: PIF and EFSA variable names

PIF Variable	EFSA Variable
Bacillus	Bacillus cereus
Campy	Campylobacter
Clostridium	Clostridium perfringens
Listeria	Listeria monocytogenes
Salmonella	Salmonella
Staphy	Staphylococcus aureus
STEC	Shiga toxin-producing Escherichia coli (STEC)
Yersinia	Yersinia enterocolitica

4.3 Compatibility issue related to different hierarchies

Another recurrent compatibility issue noticed between the EFSA and PIF databases was regarding the structure of given objects. In this case, different from what was mentioned previous sections, only changing the name of the PIFs variables to adequate to EFSAs

ones would not fix the problem, as those variables are also represented within a different structure. The difference in structure results in different levels and sub-levels for a given set of variables.

As was shown in previous sections, the PIF application renders objects from the database into drop-down menus and input boxes in the front-end, these inputs are often nested inside the structure of the own object in the database, following a hierarchy of level and sub-levels. The pattern adopted by EFSA would often not be the same adopted by PIF regarding the structure of this set of variables, resulting in databases that have completely different ways of storing their data, in structural terms.

Figure 4.2 and Figure 4.3 show an example of this structural incompatibility. In Figure 4.2 is possible to see how the set of variables regarding *PackStatus*, are organized following a structure with only one level, that is, not having nested objects; in Figure 4.3 the structure used by EFSA has two levels, having the presence of nested objects. This compatibility issue results in a need for refactoring the structure used by PIF and adapting the system on all its fronts (back-end and front-end) to handle this new structure.

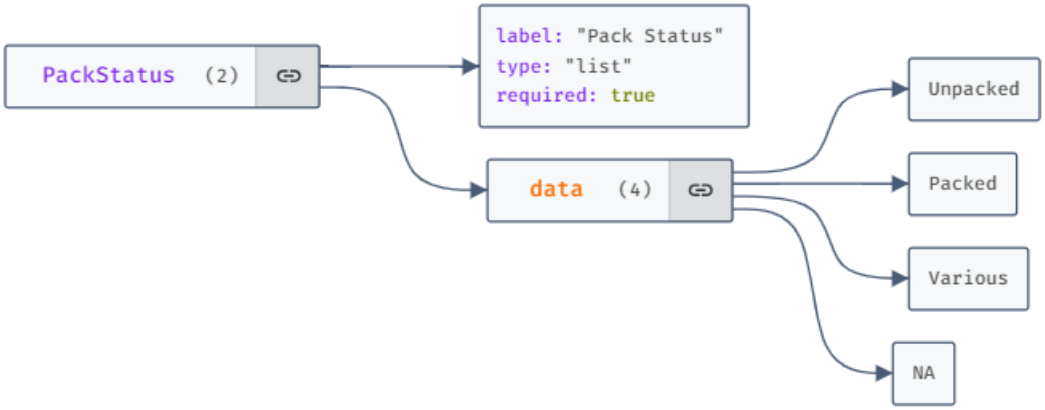

Figure 4.2: PIF *PackStatus* structure

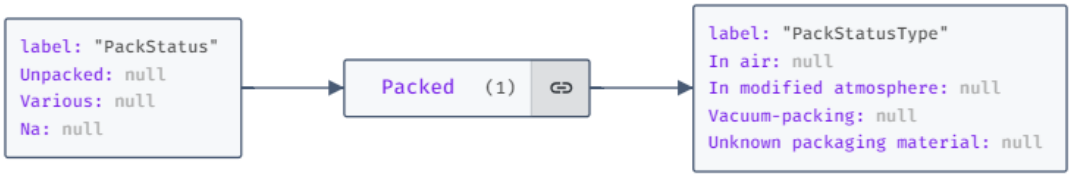

Figure 4.3: EFSA *PackStatus* structure

Some of the objects of the PIF database offered more or less flexibility, so to say, regarding changes and refactoring, the same goes for the structure implemented in the front-end and the back-end. The example depicted in Figure 4.2, shows an object that has low flexibility, given the use of arrays for rendering the options inside the drop-down menus this object is responsible for, making it difficult to add more levels to the objects.

Another relevant issue regarding the incompatibility is how changes in the database, aiming to address the incompatibility, can impact the system fronts. Using the example from Figure 4.2, the PIF front-end was designed to request and deal with the request, of an object that has an array of fields. Changing the structure of the object simply by using a nested object instead of the previous pattern, would fix the problem in the database, but would resulted immediately in a problem in the front-end.

4.4 Compatibility issue related to the codes

Every variable in the EFSA database has a unique code linked to it, in their system the user can search both for the desired name or the code related to this desired name. To provide compatibility between EFSA and PIF database is also necessary to provide this same feature, that is, allow the user to search in the PIF database using the EFSA codes. The complexity of approaching this problem is proportional to the complexity of the structures present in the PIF application.

Table 4.2 illustrates a set of EFSA variables and their respective codes. The problem related to the codes can be understood similarly to the one related to the structure, in a

way that only refactoring the database would not fix the problem, as the front-end and the back-end might not deal with the new proposed structure. To fully understand the dimension of this issue is necessary to understand how a solution for this issue would have to approach different aspect within the application; a solution that makes both databases compatible and at the same time allow the user to have the same functionally in the PIF application as the one present in the EFSA system.

Table 4.2: ESFA Codes

EFSA Code	EFSA Variable
RF-00000006-MCG	Bacillus cereus
RF-00000042-MCG	Campylobacter
RF-00000082-MCG	Clostridium perfringens
RF-00000251-MCG	Listeria monocytogenes
RF-00000304-MCG	Salmonella
RF-00003852-MCG	Staphylococcus aureus
RF-00000132-MCG	Shiga toxin-producing Escherichia coli (STEC)
RF-00002538-MCG	Yersinia enterocolitica

4.5 Proposed solution

4.5.1 Document structure diagrams

According to [21] the database per se operates more as warehouses where data related to specific things can be stored. In this sense, the use of document structure diagrams or visual data models plays an important role in the understanding of a database, they provide a graphical representation that makes it easier to understand the behavior behind the entities of a database.

Usually, the approach used for document structure diagrams involves first conceptualizing the structure and content of a database and then building it, in this sense, the approach used in this thesis worked differently. The PIF and EFSA data were already built, thus the use of the diagrams was mainly to provide additional insights regarding the entities of each database and their relationships. Fully understanding the behavior of a database is an important step before implementing any sort of migration or restructuring.

4.5.2 Data migration

Considering the challenges described in the previous sections, it becomes clear that there is a requirement for data migration. To begin with, it is important to understand the definition of data migration, as explained by [11]. The process of data migration can be understood as the preparation, transformation, and permanent movement of data to a certain stage, this process also entails the permanent shutting down of the old storage system. Furthermore, the data migration is aimed to be a one-off movement, being an action that is supposed to happen once and not regularly, similar to a one-way trip with no return [11].

Nowadays there are countless tools capable of performing the data migration process. Keeping in mind that the PIF application works with the non-relational database MongoDB gives a broad range of suitable solutions for the given problem. Using the interface provided by the Mongo Compass is possible to interact and migrate data, but the interface per se does not provide good support regarding bigger changes in a database structure.

Understanding the scenario described is fundamental to thinking about a solution that can work with MongoDB but with a more robust interface to provide more support regarding data migrations. One of the most widely programming languages used nowadays, python, provides a friendly interaction with a non-relational database and an environment for the development of migrations scripts. With the use of Python, it is also possible to create a ground structure for the entities that will be manipulated in the data migration, given a more organized and easy-to-read migration script.

4.5.3 Compatibility system

The concept of developing a compatibility system aims to address potential issues that may remain unresolved through data migration. Considering the conditions at hand, even with the use of data migration, some changes can be beyond the possible scope of an application, requiring a complete rebuild of it to fit a given requirement. In this sense, rebuilding an application to fit a change requirement might not be the best approach regarding feasibility and time, making the compatibility system an interesting approach.

Chapter 5

Development

Firstly, before any process of data migration is carried on, is important to understand how the database changes were agreed between EFSA and PIF. The team of researchers responsible for thinking about PIF structures and variables would consume the EFSA system, understand their pattern, and then elaborate on a solution that would enable the PIF application to be adequate for the EFSA system. The EFSA system works with many catalogs for displaying the data, each catalog with a slightly different hierarchy regarding their variables, in this first stage, the PIF researcher would choose the catalog with better changes of match and then elaborate a protocol to present to EFSA.

Once the protocol was elaborated, the PIF team would have a meeting with the EFSA members to present the new proposal for changes, once this proposal was approved by EFSA, the process of data migration could then be finally carried out. This process of elaborating a protocol and having a meeting to validate it would be repeated throughout the execution of the project till the PIF database was completed and migrated to the EFSA standards. As an integrated of the PIF team, I also participated in the meetings and showed the results regarding the changing progress while they were being executed.

The Annex A shows one of the protocols elaborated by the PIF team to be presented in the EFSA meeting. In this protocol, some other matters are also negotiated, as the terms conventions related to papers search, in a later section is possible to see the proposed change regarding the database. One example of the annex is the variable Food

Origin which to meet EFSA requirements should be renamed to *Country of Origin*; other than that, the EFSA also demanded to add a new variable associated called Country of Sampling. Often the change in a variable name would result in its change within their structure but also within each one of the previous records, at the point of executing this master thesis, the PIF database had around six thousand records already inserted, making it not viable to change all data manually.

5.1 Document structure diagrams

Despite the literature review carried out and the initial idea of using ontologies, they were not used during the development of the project. This was mainly because the new structure for the PIF database was not developed by this thesis, but rather extensively by PIF researchers on their own. Thus, in this context, the work of this thesis was to understand the new structure and then make the necessary changes.

In this sense, document structure diagrams were used extensively to facilitate understanding of the new and old structures of the PIF database. Although these graphical representations are not ontologies, as they do not contain the properties or roles component within the three fundamental components of an ontology, they played an important role in carrying out the data migration process. The graphical representations in question, both those developed by the PIF researchers and those developed in the body of this thesis, were fundamental for a better understanding of the database and the changes that would follow with its structures.

Throughout the data migration process, graphical representations played a significant role. They were used to initially comprehend the required changes and subsequently formulate change proposals. Figure 5.1 shows an graphical representations formulated to display the EFSA legume structure, with data making more clear how EFSA structure regarding legume works. This same process was reacted to all food-related structures and also to the migration part involving structure refactoring.

As shown in Figure 5.1, the created graphical representations used a system of colors

to identify the inner levels of the structure and make it easier to match the same levels and the new proposed structure. In this sense, the orange color was used to represent the outermost level, called in the PIF database by *Category*; the yellow color to represent the subsequent level after it, called *SubCategory*; the blue color to represent the next subsequent level called *FoodClass* and, the green color to represent the last level called *FoodSubClass*. During the process, it was decided to limit the use of just 4 levels to facilitate the process of conceptualizing the structures.

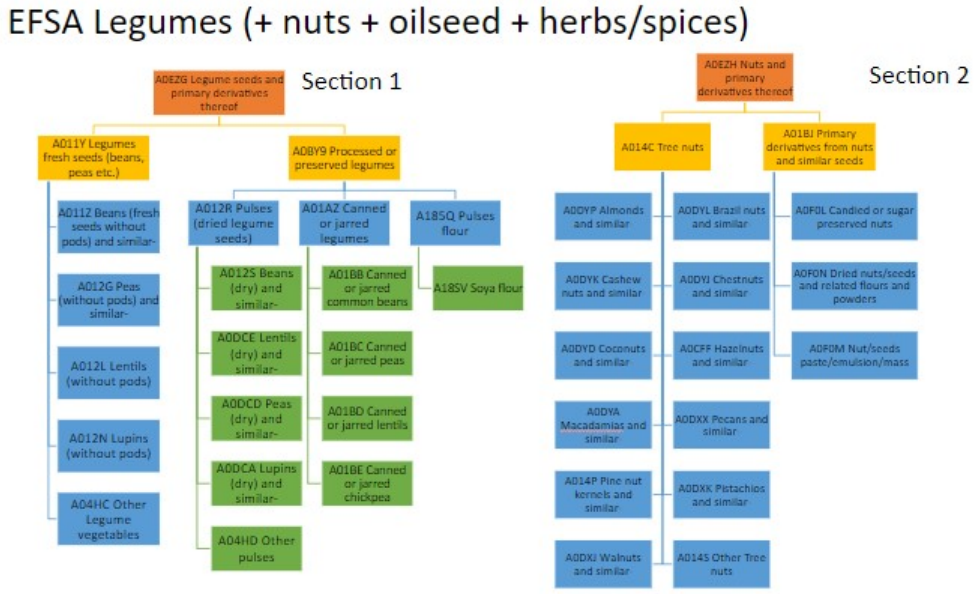

Figure 5.1: EFSA legume structure graphical representation

There was a large number of catalogs, it was up to researchers from the PIF team to analyze the different catalogs with different structures and come up with an ideal structure that could be implemented, once this structure was thought out, it was discussed in a meeting and then given - the migration process from the old structure to the new structure begins. Figure 5.2 shows the proposed structure for legumes. Another worth noticing is the division of the structure into sections, given the enormous side of the structure as a whole, making it easier to formulate changes and discuss them.

Furthermore, after the process of the proposal of a new structure, the process of implementation of the new structure also relies on the use of an representations to display

the JSON object present in the database graphically. By using that was possible to have a better understanding of the structures that would be changed and their relationship with other structures in the database. This process held particular significance because, unlike the graphical representation depicted in Figure 5.1 and Figure X1, the graphical representations developed during the implementation phase and utilized throughout the thesis were constructed from JSON objects.

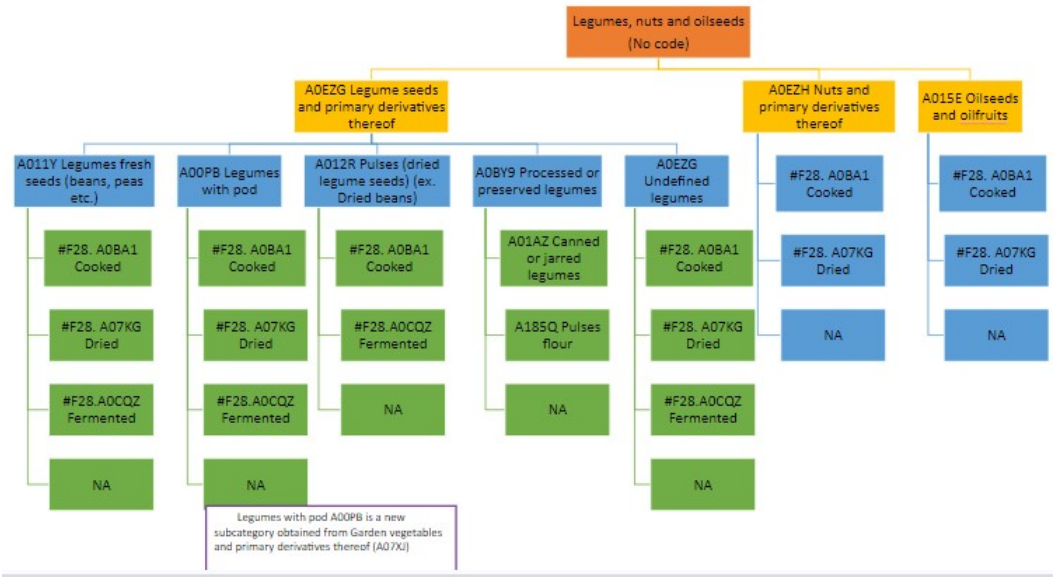

Figure 5.2: Graphical representations for new structure of Legumes

5.2 Data migration

5.2.1 Migration script

Once the agreement between EFSA and PIF was already set, was possible to start the database migration. For a great part of the process, a script of migration was developed to be executed and migrate all the PIF databases into an EFSA standard. First of all, it is relevant to understand the structure of the script responsible for the database migrations and then later its function.

Aiming to promote a greater degree of understanding, the script was developed following a class model, where each one of the collections in the database would have a respective class in the script. Listing 5.1 illustrated the class responsible for the collection *bacteria*, the methods and functions implemented by this class are mirrored by the classes related to *virus* and *parasite*. Analyzing Listing 5.1 is possible to see the two methods present in this class, when one then responsible for updating keys in the collection and another one is responsible for updating values.

In the initial stages of code development, it was a challenging to anticipate the full extent of the implementations that would later be integrated into the PIF database, once the changes were being agreed in the process of development of the thesis. Keeping that in mind, the class structure of the migration script was thought to allow for changes and further development. In a later stage of the development of the script, all the classes were rethought and reformulated to allow a more generic solution that could be also applied in different databases with different scenarios.

Listing 5.1: Bacteria class

```
class Bacteria:
    def __init__(self, uri, database): ...

    def update_value_collection(self, key, value, new_value):
        if value == new_value:
            return
        self.db_collection.update_many(
            {key: value},
            {"$set": {key: new_value}})

    def update_key_collection(self, key, new_key):
        if key == new_key:
```

```
            return
        self.db_collection.update_many(
            {},
            {"$rename": {key: new_key}})
```

Following the same class model, the structural collections in the database also have a corresponding class in the migration script. In the first stage of development, the classes responsible for the three main structural collections (bacterial structure, parasite structure, and virus structure) were separated into three different classes; in a future review, the three classes could be represented using only one generic class.

Listing 5.2 shows the code implemented for the class *BacteriaStruct*. In the evolution of the script, more methods were implemented in this class to allow it to handle new requirements. Overall, the change of a variable was executed using the structural class, to change the way the variable is written in the database, and then the non-structural class, to change all the records in retrospect that had the variable old name still.

Listing 5.2: Bacteria Structure class

```
class BacteriaStruct:
    def __init__(self, uri, database): ...

    def update_key_collection(self, object_id, key, new_key):

        self.db_collection.update_one(
            {"_id": object_id},
            {"$rename": {key: new_key}})

    def insert_key_collection(self, object_id, key,
    key_value=None):
        object_id = ObjectId(object_id)
```

```
        self.db_collection.update_one(
            {"_id": object_id},
            {"$set": {key: key_value}})

    def remove_key_collection(self, object_id, key):
        object_id = ObjectId(object_id)
        self.db_collection.update_one(
            {"_id": object_id},
            {"$unset": {key: ""}})
```

Listing 5.3 shows the first lines of the migration script. Analyzing Listing 5.3 makes it possible to see how the classes related to bacteria and bacteria structure are instantiated. Another relevant factor is that the implemented classes have a constructor that asks for two parameters: *uri* and *database*. This feature was especially useful through the development of the script to allow a quick change in the database that the script would run; changing the variable *database* would make the script run a different local database.

Listing 5.3: Class instantiation

```
uri = "mongodb://localhost:27017"
database = "pathogensDB"

bacteria_struct = BacteriaStruct(uri, database)
bacteria = Bacteria(uri, database)
```

Another noteworthy aspect, is the creation of variables responsible for the called "trees", as depicted in Listing 5.4. They showed that "trees" are records inside the structural collections of the database, these records were manipulated when a variable needed to be changed or a new variable needed to be added or removed; in later development, some of the records (trees) also need to be completed remodeled to adapt to new requirements.

Listing 5.4: Database trees

```
bacteria_main_tree = '5f54602de9704b205cbdbe06'
bacteria_general_results = '5f315d7c3d418525e0385fa3'
bacterias = '5f3160623d418525e0385fa8'
```

Finally, Listing 5.5 shows how the migration script was developed using the methods inside the created classes. In the first line the method *update value collection*, from the *Bacteria* class, is used to change the value of the "Bacterium" variable in all the records in the database. In this specific case, was only relevant to change the value inside the variable *Bacterium* from *Bacillus* to *Bacillus cereus*, not being necessary to change its key.

Different from the methods in the non-structural classes, the methods inside the structural classes always need additional parameters: the *tree*. In the non-structural classes, all methods were designed to be executed on all the records within their respective collections in the database; whereas within the structural classes, the methods were created to update individual records (trees).

In Listing 5.5 it is evident the working of the *update key collection* method. In this case, the variable *FoodOrigin* will be renamed to *CountryOfSampling* but only within the *bacteria main tree* which is the most important record inside the bacteria structural collection in the database.

It is worth noting that the need for changing the variable key or value was intrinsically linked to the way the PIF application was developed. In some cases the information shown to the user was collected from the variable value in some other cases was collected from the variable key.

Listing 5.5: Class methods

```
bacteria.update_value_collection(
    "Bacterium",
```

```
    "Bacillus",
    "Bacillus cereus"
)

bacteria_struct.update_key_collection(
    bacteria_main_tree,
    "FoodOrigin",
    "CountryOfSampling"
)
```

After understanding the migration script, it is also important to understand the technical part of the process. In the migration process, the script was executed in a local version of the database and then uploaded directly to the server. Before being updated to the server the new version of the database underwent testing in a local version of the back-end and the front-end of the PIF application.

In the advanced stages of development, the migration script and its associated classes underwent a refactoring process. Listing 5.6 shows how the refactoring process happened. The first point to be noted is the class *MultiRecordUpdater* which is a generic class used to substitute *Bacteria*, *Virus* and *Parasite* classes; within this new class is possible to choose the collection used by the class while instantiating the class, passing an additional parameter in class constructor.

Listing 5.6: Migration script refactoring

```
with open("bacteria_updates.json", "r") as json_file:
    bacteria_updates = json.load(json_file)

bacteria = MultiRecordUpdater(uri, database, "bacteria")

for key, value, new_value in bacteria_updates:
```

```
bacteria.update_value_collection(key, value, new_value)
```

Another worth noticing point is the use of JSON files and loops in the new version of the migration script. The idea behind such use is to prevent the repetition of lines and provide a more clean code. In the example illustrated in Listing 5.6, the method *update value collection* is called inside the for loop and executed for all the lines inside the JSON object. Listing 5.7 shows what the JSON object used in the previous example looks like.

Listing 5.7: Json for the migration script

```
[
    ["Bacterium", "Bacillus", "Bacillus cereus"],
    ["Bacterium", "Campy", "Campylobacter"],
    ["Bacterium", "Clostridium", "Clostridium perfringens"],
    ...
]
```

5.2.2 Side scripts

Despite the effectiveness of the migration script for a great part of the migration process, the same could not be used when it came to migrating the structural food collection in the database. For this specific migration was necessary not only to change, add, and remove variables but also to refactor the structure inside the collection. In this way the migration script was executed mainly to the collections related to the agents (structural and non-structural), and sides scrips were developed to deal with the migration of the food-related variables.

The process for migrating the food-related collection was done in two parts, the first one was the manual refactoring of the old structure; the second part was the update of the recorded data in the database to meet the new structure. Before looking at how the scripts were developed, it is important to understand how these changes were accorded

with between PIF and EFSA, and how the changes should be carried out.

Figure 5.3 depicts a section of the new proposed structure to meet EFSA requirements. Once the researcher analyzed the old PIF structure and what would be necessary to change to meet the requirements, they would create a document that would depict the new structure so that could be shown in a meeting and approved for changes. Once this change was approved the migration process could then move on.

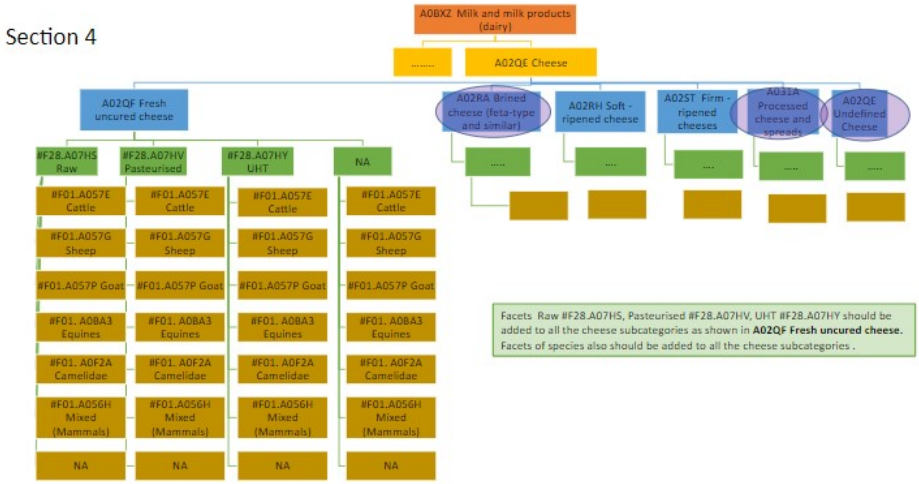

Figure 5.3: New proposed structure

Figure 5.4 shows the migration rules related to the record data in the PIF database; upon these rules, the side scripts could be developed to make the stored data meet the new structure. Listing 5.8 illustrates how the side scrip was developed to meet the migration rules shown in Figure 5.3.

PIF food Category		EFSA Terminology		Observations/ Suggestions
		Code	Name	
Category	Dairy	A0BXZ	Milk and milk products (dairy)	
	Subcategory=Cheese Label= `Fresh`	A0BXZ A02QE A02QF	→Milk and milk products (dairy) →Cheese →Fresh uncured cheese	Notice that for CHEESE, HeatTreatedCheese (level 4) and FoodClassSpecie (level 5) can remain the same.
	Subcategory=Cheese Label= `Feta`	A0BXZ A02QE A02RA	→Milk and milk products (dairy) →Cheese →Brined cheese (feta-type and similar)	

Figure 5.4: Migration rules

It is important to understand, that at this point in the migration process, the new structures added a much broader range of categories and subcategories within the trees,

making it sometimes impossible to migrate the old stored data to the new proposed form directly. In this way, the variable *label* was used extensively, as this variable contains information related to the inserted record that could be used in the data migration.

The drawback associated with this variable was that it was inserted by an open field in the front end, making it necessary to create a regex pattern to read this data and change the record. Often the value of the variable *label* would have grammatical variations of writing, making it necessary to formulate a new regex for each subcategory inside a given category in the food structure.

Listing 5.8: Food related migration script

```
for document in collection.find(
    {"Category": "Dairy",
     "SubCategory": "Cheese",
     "Label": {"$regex": "fresh", "$options": "i"}}):
    old_food_sub_class = document.get("FoodSubClass")
    old_food_class_specie = document.get("FoodClassSpecie")

    collection.update_one({"_id": document["_id"]}, {"$set": {
        "Category": "Milk and milk products (dairy)",
        "SubCategory": "Cheese",
        "FoodClass": "Fresh uncured cheese",
        "FoodSubClass": old_food_sub_class or "NA",
        "FoodClassSpecie": old_food_class_specie or "NA"
    }})
```

5.2.3 Manual migration

As each structure behaved differently from the other and as the changes were not made as a whole, involving all structures at once, but rather progressively, it was very difficult

to propose a script to solve this problem. In this case, the new structures were rewritten using a text editor to help correct possible errors with the JSON object and re-inserted into the database manually; once inserted, the structures were tested with the local versions of the back-end and front-end for later uploading to the server.

After approval of the proposed new structure by EFSA, the process of writing the new structure could begin. Figure 5.5 and Figure 5.6 show a fragment, in the new and old format, of the structure related to dairy. It is possible to understand that both structures remained with four levels, but the organization of these levels was completely remodeled.

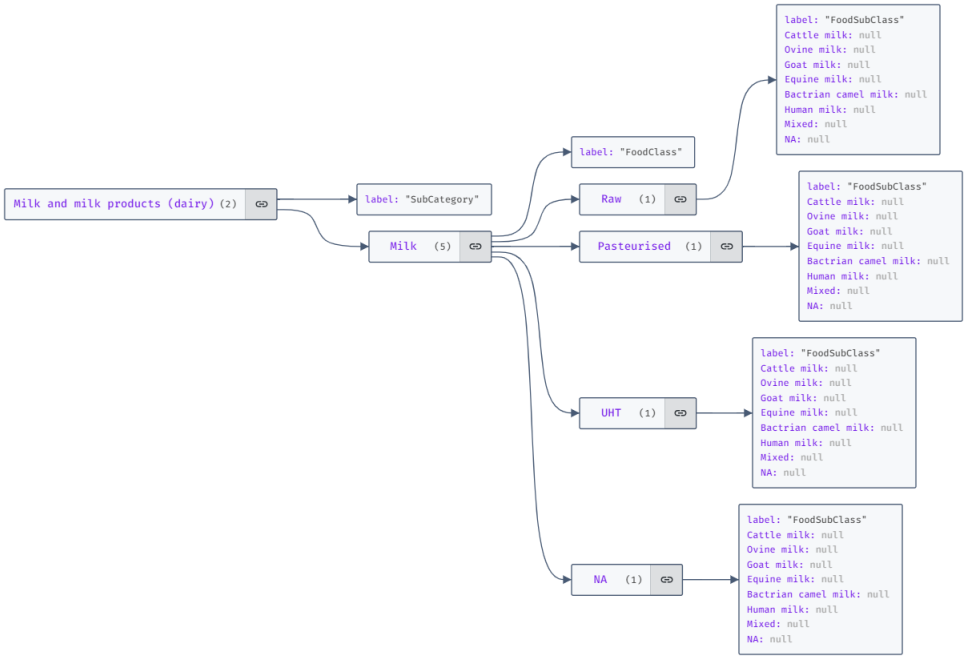

Figure 5.5: New dairy structure

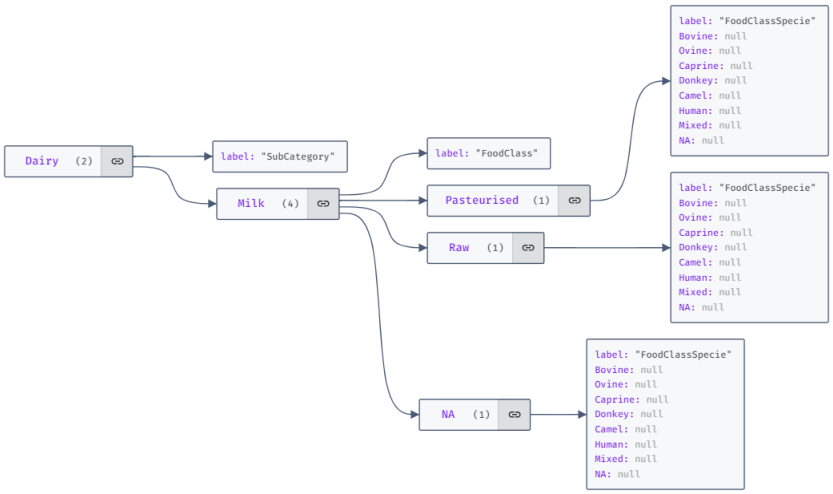

Figure 5.6: Old dairy structure

5.3 Compatibility system

The compatibility system was created to address the issue related to the EFSA codes. Each variable in the EFSA system has also a respective code associated with it, which would make it necessary for the PIF application to incorporate the code system to provide a full migration with the EFSA system. To approach this problem a new page was created in the PIF application that allows the user to perform searches using the EFSA codes.

It is important to mention that the changes mentioned in this section were also accorded with EFSA; so it was requested from them that the search by their codes should be possible within the PIF application. Having that in mind, the proposed solution aims to fulfill the EFSA requirements and have the minimum impact on the database.

One possible solution for the EFSA code issue would be the creation of a new column for each code associated with a variable, this solution would duplicate the number of columns in the table of the PIF application. Considering the significant impact this change would have on the database, the PIF team looked for a solution that could effectively address the issue while minimizing its impact on the database.

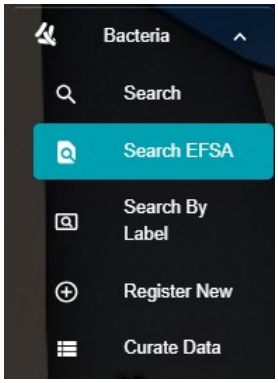

Figure 5.7: PIF navigation bar

Figure 5.7 shows the option in the navigation bar responsible for the compatibility system caller *Search EFSA*. Every one of the agents (bacteria, parasite, virus) in the PIF application would have its own *Search EFSA* page, following the decentralized logic behind the application. The behavior associated with the *Search EFSA* page is similar to the behavior of the *Search* page, present in the PIF application, already explained. Each one of the options illustrated in Figure 5.7 executed actions only in the collections respective to their agents.

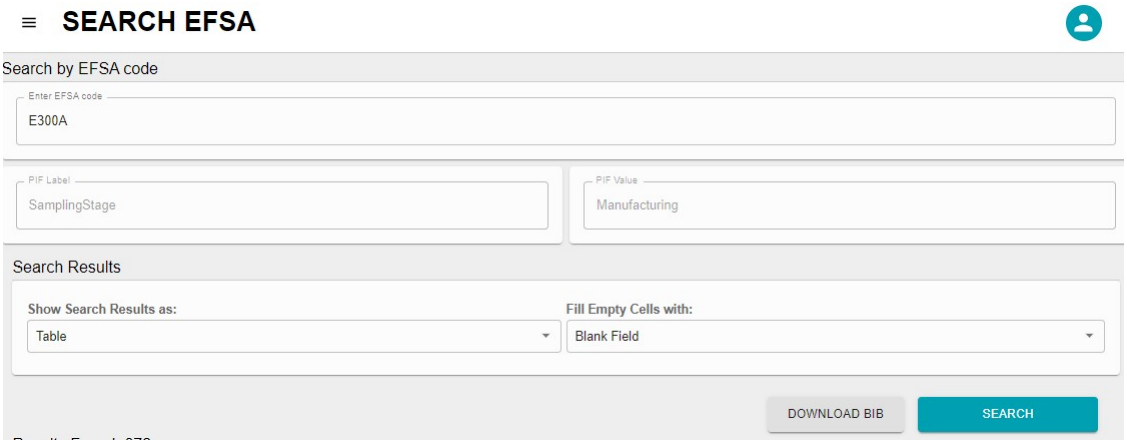

Figure 5.8: Search EFSA page

Figure 5.8 depicts the interface of the *Search EFSA* page. In Figure 5.8 is possible to understand how the system works: the user inserts a code in the *Enter EFSA code* box and then presses the button search; the search will return in the boxes *PIF Label* and

PIF Value the respective variable for the given code and then the table with the values related to the searched code are rendered. Figure 5.9 shows how the query related to the given code is sent to the back-end.

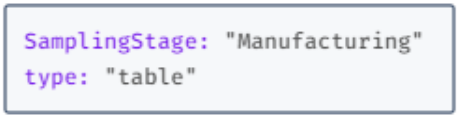

Figure 5.9: Search EFSA page query

Figure 5.10: Compatibility object array

It is worth noticing in Figure 5.9 that the query executed by the back-end in the database is always with the PIF term and not with the actual code, this feature is given exactly thanks to the compatibility system that executes the conversion of the EFSA

code into a PIF term. Figure 5.10 shows the array of objects behind the compatibility system, in this context is fundamental to understand the behavior of this array and the code behind its performance.

Noteworthy in Figure 5.10 is how the array of code is structured. The array is composed of objects, and each object is responsible for mapping a given variable in the PIF application with the related EFSA codes. As an example, the object with the *Label SamplingStage* is responsible for mapping the PIF variable *SamplingStage*; this variable can assume three different values in the PIF application: *PrimaryProduction*, *Manufacturing* or *Distribution*, each one of those possible values are mapped to different codes.

The solution proposed for the compatibility system is based on a JavaScript pattern called *Lookup Tables*. As stated by Zakas (2010) the use of this pattern provides a faster and easier-to-read code, especially in cases where the use of if-else or switch statements would result in an immense number of conditionals. In this context, using an array of objects to deal with conditional works better and faster than any other method.

According to Myalapalli and Karri (2015), the use of the *Lookup Tables* pattern can also be understood as an optimizing tool in the front-end of the application. In accordance with Myalapalli and Karri (2015) the use of *Lookup Tables* is particularly effective when assigning a single key to a single value, which is precisely the solution approached by the compatibility system.

Listing 5.9 shows the function responsible for implementing the logic of the *Lookup Tables* in the front end of the application. As is notable in Listing 5.9, the function iterates over the *codes* array, providing the value corresponding to the key. Listing 5.10 shows the *useState* responsible for receiving the *codes* array, once the *Seach EFSA* page is loaded, thanks to the use of a *useEffect*, the back-end API is called and the *codes* array is populated; with the *codes* array already populated, any time the user use the search button, the iterating with the objects inside the array can happen.

Listing 5.9: The handleCodeEFSA function

```
const handleCodeEFSA = (data) => {
      for (let i = 0; i < codes.length; i++) {
        const codeObject = codes[i];
        const codeObjectLabel = codeObject.Label;

        if (codeObject.hasOwnProperty(data)) {
          return {
            [codeObjectLabel]: codeObject[data],
            type: searchType
          };
        }
      }

      return { type: searchType };
    };
```

Listing 5.10: Codes useState

```
const [codes, setCodes] = useState([]);
```

5.3.1 Compatibility system side implementation

The logic behind implementing the *Lookup tables* pattern in the compatibility system could extend its applicability to resolving minor issues in other areas of the data migration process. For the data migration of the the food-related variables, the structure responsible for the *Aquatic based food* needed to be refactored; in the refactoring process two new variables need to be inserted in the new structure, both related to the same subject.

In accordance with the PIF team, it was decided to only insert one variable in the database, and then map its value to another matching variable. The variables in question

relate to the common and scientific names of aquatic animals, Figure 5.11 shows part of the object responsible for performing the match between the scientific name and the common name of a given aquatic animal. In this approach, only the scientific name related to the given animal is inserted in the database.

In this context, it is important to understand the motivation behind this present implementation. In an analysis by the PIF was understood that every study that is read for future insertion in the database always has their animals given by the scientific name, consequently, it would be reasonable to only insert the scientific name in the database. Nonetheless, as the PIF application is open for new users, the use only of the scientific name could result in the struggle of a non-trained person in searching data.

In this way, as shown in Figure 5.12, a compatibility box was taught to give the user a hint regarding the animal's common name. The *Common Name* box depicted in Figure 5.12 has a conditional rendering associated with the *Aquatic based food* category and its inner level *FoodClassSpecie*, in this way, this box is only shown in the front-end for this specific case, as required by implementation requirements. Once the compatibility box is rendered, for each value selected by the user for *FoodClassSpecie* a matching common name will be displayed in the *Common Name* box.

```
Eriocheir sinensis: "Chinese mitten crab"
Procambarus clarkii: "Red swamp crawfish"
Astacus leptodactylus: "Danube crayfish"
Macrobrachium rosenbergii: "Giant river prawn"
Macrobrachium malcolmsonii: "Monsoon river prawn"
Macrobrachium nipponense: "Oriental river prawn"
Exopalaemon modestus: "Siberian prawn"
```

Figure 5.11: Compatibility object

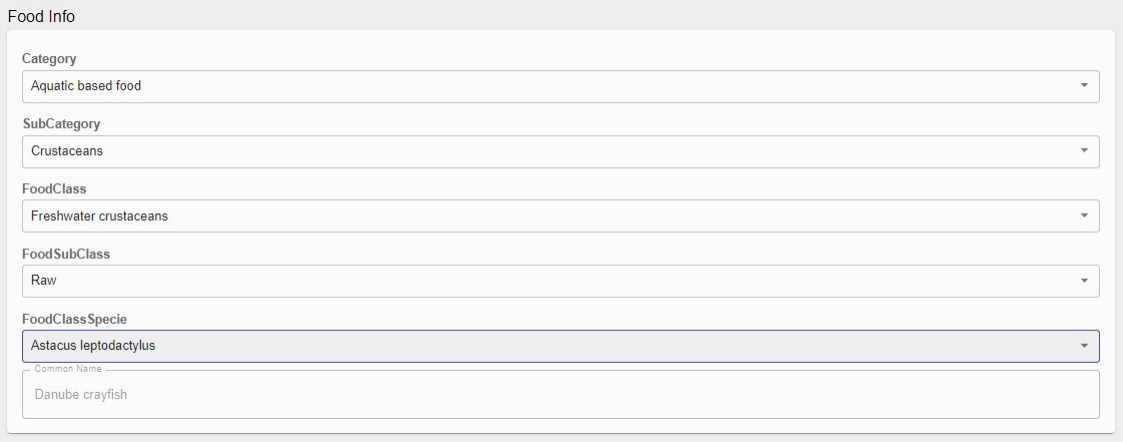

Figure 5.12: Compatibility box

Chapter 6

Results

It's important to understand that, although within the scope of the thesis, this chapter is reserved for the discussion and testing of the results developed in the thesis, the approach within the scope of this thesis took place somewhat differently. The results of the migration process carried out by this thesis were gradually executed, exposed, and tested by both the PIF team and the EFSA group during the development of the thesis itself. These iterations validate the capability of the proposed solution to address the requested problems. Within the scope of thesis execution, one can then conceive a methodology for the migration of a database composed of three levels: *Understanding of structures*, *Formulation of the migration script*, and *Compatibility system*.

As a result of these efforts, all data in the PIF database has been successfully transformed to adhere to the EFSA standard. This migration encompassed a total of 5.852 records, including 5.924 records related to bacteria, 303 records related to parasites, and 625 records related to viruses, aligning with the compatibility proposal's requirements.

6.1 Understanding of structures

Analyzing the execution of the thesis makes it clear the importance behind understanding the structures of both databases before the data migration process. On one hand, understanding the structures of a database, even in isolation without considering any migration,

is already relevant within the comprehension of an application. The structures present in a database can be seen as the foundations of an application, so their understanding also provides a much deeper insight into the overall functioning of the application, both in the back-end and the front-end.

From another perspective, understanding the structures of a different database that addresses the same subject promotes insights into the organization of your own database. The process of analyzing the structure of an external database, especially in the case of EFSA, where there were various different structures (due to the use of different catalogs), and determining which structure would be more suitable for matching with your database, is a process that creates insights and broadens knowledge. Finally, in a last analysis, rethinking the original structure of your database to enable the match with a new structure, following the constraints established by the migration process, such as the limit of four levels for the new PIF structures, also provides powerful insights.

6.2 Formulation of the migration script

The subsequent step after understanding the structures is actually carrying out the migration of these structures. In this context, the classes created and the execution structure used in this thesis offer the possibility of implementation in other databases. With the refactoring of the migration script, as described in previous chapters, it now exhibits a generic behavior that can be executed in other databases with only changes to the parameters used for the class methods.

In Annex A, where the migration script is described, it is possible to notice two classes: *MultiRecordUpdater* and *SingleRecordUpdater*. By using these two classes, it is possible, with the use of a Python script, to manipulate any variables within a MongoDB database. The *MultiRecordUpdater* class operates with the *updateMany* method present in MongoDB, performing updates on multiple documents at once. On the other hand, the *SingleRecordUpdater* class operates with the MongoDB *updateOne* method, updating one document at a time.

Given the creation of the writing classes, the migration script proposed by this thesis also offers ways to use the methods of these classes to reduce code repetition and increase readability. The class methods provided by *MultiRecordUpdater* and *SingleRecordUpdater* can be used in conjunction with loops and external JSON files. This solution allows a method to be written only once, and its parameters are consumed from the external JSON file and changed as the loop is executed, making each line of the external JSON file an execution for the given method. Another execution approach present within the migration script (Annex B) is the use of functions to update different entities with common values.

6.3 Compatibility System

The solution proposed by the compatibility system offers a viable option for converting parameters between databases without undergoing any actual data migration process; it functions as a conversion system. This solution is applicable not only in the context of data migration but also in the context of application enhancement, where changing the database may not be a feasible option. In the development of the compatibility system for this thesis, as explained in a previous section, the development was based on what was requested by EFSA and what was agreed upon among the PIF members.

In this sense, the compatibility system can be understood as the final phase in the process of converting the PIF database into the patterns imposed by EFSA. The system enables the PIF application to function as intended, meeting its own objectives, and also provides additional features to align with EFSA standards. A user who accesses the EFSA system and can search for terms by code can perform the same search in the PIF system using the same codes and obtain results with the same nomenclature as the EFSA standard, thanks to a collaborative solution between the migration script and the compatibility system.

6.4 Addressed Issues

In this context, it is possible to understand how the three compatibility issues described, namely, *Compatibility issue related to different variable names*, *Compatibility issue related to different hierarchy*, and *Compatibility issue related to the codes*, were tackled by the thesis. However, they sometimes deviated from the initial objectives proposed. In the proposal stage of the thesis, it was expected that the execution of the thesis work would be done in stages. Still, what happened in reality was a continuous execution of small changes throughout the thesis work.

Within the scope of the executed changes, they were conceived during the execution of the thesis, meaning that the data migration proposals did not cover the entire application at the beginning of the thesis. During the months of thesis execution, proposals for changes to the database were created and subsequently discussed in meetings with EFSA. Therefore, the initial idea of using machine learning for data migration proved ineffective because the changes had to be executed within a short period, and there would be no significant sample for training. Another relevant point is that it was not possible to predict to what extent the structure of PIF should be altered to meet EFSA's requirements. Therefore, the proposed migration models (migration script and compatibility system) were created to allow scalability when necessary.

Despite the proposed migration models enabling some scalability and meeting EFSA's requirements regarding database migration, they also have limitations. In the context of a database migration where there is a general understanding of all the implementations needed from the very beginning, it is worth considering a solution that can address more problems at once. The solution presented in this thesis works well, especially for small and medium-sized migrations, where the number of variables and migrated structures is not excessive, and the object array for the compatibility system is not too long.

In a scenario with a high number of variables to migrate, as well as the need to rewrite new structures, it is worth thinking about a solution that can address both issues at once or handle a greater number of modifications with fewer lines of code. The final migration

script of the thesis consists of 167 lines of code, which would scale up quickly if the number of migrated variables were larger or if there were more entities involved.

Another point is the rewriting of data structures in the database to comply with EFSA's requirements. As explained in previous sections, there were times when it was necessary to not only change the value or key related to a variable but also its structure. Within the scope of this thesis, these changes were made manually, with the assistance of only a text editor, to correct errors in the structuring of the new JSON object. Looking at a larger scope, for a large-scale modification with a significantly larger number of structures, the approach used in this thesis would not be ideal.

Yet another point concerns the compatibility system. Given the quantity of EFSA codes and related variables, the system functions satisfactorily. However, if the number of variables and codes were much larger, the object array responsible for the matching would grow significantly. This implies two evident problems: slower processing by the compatibility system due to a very large array and difficulties in maintaining and correcting this array in the future. For this reason, it is thought that in a system that requires a very large number of code-variable relationships, direct insertion of new columns into the database may be a better solution, aiming to facilitate maintenance and improve performance.

Chapter 7

Conclusion and Future Work

In conclusion, this thesis addressed the migration of the PIF database to the standard adopted by EFSA, aiming to make the PIF application fully compatible with EFSA's. Within the scope of this thesis, compatibility issues between the databases were comprehensively addressed, resulting in a new version not only of the PIF database but also of the entire application. The development of the new version of the PIF database and the application as a whole not only met the established migration requirements but also provided a better understanding of the PIF application in all its facets: front-end, back-end, and the database.

In the context of enabling future changes if necessary or another migration process, the developed script is capable of handling such migrations, provided they are on a similar scale to that undertaken within this thesis. It is understood that the compatibility system developed also allows for scalability. These scalability features are crucial in modern application development solutions, given the rapidly changing demands of applications in today's landscape. Additionally, in case EFSA's adopted standards undergo changes, the developed solutions are capable of handling refactoring.

Considering the difficulties encountered during the execution of the thesis, one can think of a problem that directly affected the PIF team responsible for conceptualizing the new structures, especially those related to food. It is understood that, given the literature review in the field of ontologies for food, although there are indeed well-known and widely

popularized standards, there is still no mandatory standard to be followed. Thus, there is still room for disagreement among researchers regarding the structuring of a unified ontology for food. In this context, thinking about an unified ontology that can be used as a universal standard would be of great relevance within the field.

Within the spectrum of ontologies, we can also think of many other future applications. Even after the data migration process and compatibility between the PIF and EFSA databases, it is still possible to think about the process of improving the structures used for future refactoring. In this context, the use of ontologies can play an important role in the process of rethinking the current structures and proposing changes that in some way improve the current context even further.

Another relevant topic for future work would be the use of machine learning. During the thesis, a great deal of information was gathered about both the PIF and EFSA databases, and it was possible to gain a better understanding of how the PIF database works, how the entities in it are related, which are the most important variables and collections in the database, as well as various other pieces of information. In this sense, this information would be of great value to machine learning algorithms for a wide variety of applications.

Thinking about possible applications for the use of machine learning within the information obtained, it is possible to think about the use of algorithms to analyze the structures present in the PIF database in order to propose possible improvements. Other relevant data are the records already entered in the database relating to food-borne pathogens. This data opens doors not only for the use of machine learning but also for the use of data analysis.

Still, within the area of machine learning, another possible application would be the use of algorithms to automate the data migration process. With the current knowledge of the structures involved in the PIF database and the structures in the EFSA database, as well as the knowledge of all the variables and entities that have undergone the migration process, it would be possible to think of a possible automated data migration process.

In conclusion, it can be understood that the processes and solutions described in this

thesis address the proposed problems and allow for the scalability of these solutions. However, it is recognized that the scalability of any solution is limited by its own conditions.

Bibliography

[1] H. Abbes, S. Boukettaya, and F. Gargouri, "Learning ontology from big data through mongodb databasw," *ScienceDirect*, 2015.

[2] A. Gómez-Pérez and V. R. Benjamins, "Applications of ontologies and problem solving method," *Al Magazine*, vol. 20, no. 1, pp. 119–122, 1999.

[3] J. Lehmann and J. Völker, Eds., *Perspectives on Ontology Learning*. Informatics Institute, University of Leipzig, Germany; Data & Web Science Research Group, University of Mannheim, Germany, 2014.

[4] E. Arnaud, M.-A. Laporte, S. Kim, *et al.*, "The ontologies community of practice: A cgiar initiative for big data in agrifood systems," *Patterns*, vol. 1, p. 100 105, Oct. 2020. DOI: 10.1016/j.patter.2020.100105.

[5] M. Amith, C. Onye, T. Ledoux, G. Xiong, and C. Tao, "The ontology of fast food facts: Conceptualization of nutritional fast food data for consumers and semantic web applications," in *Proceedings of The 5th International Workshop on Semantics-Powered Data Mining and Analytics (SEPDA 2020)*, Dec. 2020, pp. 1–15.

[6] D. A. Alghamdi, D. M. Dooley, G. Gosal, E. J. Griffiths, F. S. Brinkman, and W. W. Hsiao, "Foodon: A semantic ontology approach for mapping foodborne disease metadata," *BC Center for Disease Control*, pp. 1–2, 2023.

[7] M. C. Chibucos, A. E. Zweifel, J. C. Herrera, *et al.*, "An ontology for microbial phenotypes," *BMC Microbiology*, vol. 14, pp. 1–8, 2014. DOI: 10.1186/s12866-014-0294-3.

[8] S. Meng, T. Torto-Alalibo, M. C. Chibucos, B. M. Tyler, and R. A. Dean, "Common processes in pathogenesis by fungal and oomycete plant pathogens, described with gene ontology terms," *BMC Microbiology*, vol. 9, no. Suppl 1, S7, 2009. DOI: 10.1186/1471-2180-9-S1-S7.

[9] I. Mangone, N. Radomski, A. Di Pasquale, *et al.*, "Refinement of the cohesive information system towards a unified ontology of food terms for the public health organizations," *National Reference Centre (NRC) for Whole Genome Sequencing of microbial pathogens: data-base and bioinformatics analysis (GENPAT), Istituto Zooprofilattico Sperimentale dell'Abruzzo e del Molise "Giuseppe Caporale" (IZSAM)*, 2023.

[10] M. A. Mohamed, O. G. Altrafi, and M. O. Ismail, "Relational vs. nosql databases: A survey," *International Journal of Computer and Information Technology*, vol. 03, no. 03, pp. 598–601, 2014, ISSN: 2279–0764.

[11] J. Morris, *PRACTICAL DATA MIGRATION*, 2nd. Swindon, UK: BCS Learning & Development Ltd., 2012.

[12] Y. S. Wijaya and A. A. Arman, "A framework for data migration between different datastore of nosql database," *IEEE*, 2018. DOI: 10.1234/your-doi-if-available.

[13] A. Bansel, H. Gonzalez–Vélez, and A. E. Chis, "Cloud-based nosql data migration," *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 224–231, 2016. DOI: 10.1109/PDP.2016.111.

[14] A. Hillenbrand, U. Störl, S. Nabiyev, and M. Klettke, "Self-adapting data migration in the context of schema evolution in nosql databases," *Distributed and Parallel Databases*, vol. 40, no. 5-25, 2022. DOI: 10.1007/s10619-021-07334-1.

[15] M. Svihla and I. Jelinek, "Two layer mapping from database to rdf," *Journal Name or Conference Proceedings*, vol. Volume Number, Page Numbers, Year.

[16] N. Cullot, R. Ghawi, and K. Yétongnon, "Db2owl: A tool for automatic database-to-ontology mapping," *Journal Name or Conference Proceedings*, Year.

[17] J. F. Sequeda, S. H. Tirmizi, O. Corcho, and D. P. Miranker, "Direct mapping sql databases to the semantic web: A survey," *Journal Name or Conference Proceedings*, Year.

[18] V. K. Myalapalli and S. Karri, "Optimizing front end applications and javascript," *International Journal of Innovative Research in Science, Engineering and Technology*, vol. 4, no. 6, pp. 4010–4030, Jun. 2015.

[19] N. C. Zakas, *High Performance JavaScript*, 1st. United States of America: Yahoo!, Inc., 2010, ISBN: 978-0-596-80279-0.

[20] S. Ioannidou, "Efsa catalogue browser user guide," Oct. 2019.

[21] M. Sir, Z. Bradac, and P. Fiedler, "Ontology versus database," *IFAC-PapersOnLine*, vol. 48, no. 4, pp. 220–225, 2015.

Appendix A

Protocol for the Pathogens in Foods

Systematic Review Protocol for the "Pathogens in Foods" Database: Prevalence and Concentration of Main Biological Hazards in Food Matrices

Contents

1. Main Objective of the Systematic Review 2
2. Protocol for the Systematic Review 3
2.1 Review question 3
2.2 Criteria for inclusion of individual studies 4
2.3 Searching for individual studies 5
2.4 Deduplication of individual studies 8
2.5 Selecting the individual studies 9
2.6 Assessing the methodological quality 9
2.7 Data extraction from primary research studies 10
2.7.1 General framing data 11
2.7.2 Study characteristics data 11
2.7.2.1 Study characteristics for bacteria 14
2.7.2.2 Study characteristics for viruses 19
2.7.2.3 Study characteristics for parasites 21
2.7.3 Outcome data 24
2.7.3.1 Outcome data for bacteria 24

Appendix B

Migrations script

Listing B.1: Main migration script

```
from multiRecordUpdater import MultiRecordUpdater
from singleRecordUpdater import SingleRecordUpdater
import json


uri = "mongodb://localhost:27017"
database = "pathogensDB"

"""
            bacteria
"""
with open("bacteria_updates.json", "r") as json_file:
    bacteria_updates = json.load(json_file)

bacteria = MultiRecordUpdater(uri, database, "bacteria")

for key, value, new_value in bacteria_updates:
```

```
18      bacteria.update_value_collection(key, value, new_value)
19
20  """
21              parasite
22  """
23  parasite = MultiRecordUpdater(uri, database, "parasite")
24
25  """
26              virus
27  """
28  virus = MultiRecordUpdater(uri, database, "virus")
29
30  virus.update_value_collection("Virus", "Hepatitis␣A␣(HAV)", "
      Hepatovirus␣A")
31  virus.update_value_collection("Virus", "Hepatitis␣E␣(HEV)", "
      Orthohepevirus␣A")
32  virus.update_value_collection("Count.EssayEnum", "ISO␣
      15216:2017", "ISO␣15216-1:2017␣Hepatitis␣A␣virus␣and␣
      norovirus")
33  virus.update_value_collection("Prevalence.EssayDet", "ISO/TS␣
      15216:2013", "ISO␣15216-2:2013␣Hepatitis␣A␣virus␣and␣
      norovirus")
34  virus.update_value_collection("Prevalence.EssayDet", "ISO␣
      15216:2019", "ISO␣15216-2:2019␣(Hepatitis␣A␣&␣norovirus)")
35
36  """
37              common update different agents
38  """
39
```

```
40 def common_update_different_agents(updater):
41     updater.update_value_collection("Stage", "Farm", "Farm")
42     updater.update_value_collection("Stage", "MidProcessing",
           "Manufacturing")
43     updater.update_value_collection("Stage", "EndProcessing",
           "Storage")
44     updater.update_value_collection("Stage", "Retail", "
       Retail")
45     updater.update_value_collection("Stage", "Restauration",
       "Restaurant")
46     updater.update_key_collection("FoodOrigin", "
       CountryOfSampling")
47     updater.update_key_collection("Stage", "SamplingStage")
48     updater.update_key_collection("NSamples", "
       TotalUnitsTested")
49     updater.update_key_collection("HeatTreatedCheese", "
       FoodSubClass")
50
51 common_update_different_agents(bacteria)
52 common_update_different_agents(parasite)
53 common_update_different_agents(virus)
54
55 """
56             bacteria_struct
57 """
58 bacteria_struct = SingleRecordUpdater(uri, database, "
   bacteria_struct")
59
60 bacteria_main_tree = "5f54602de9704b205cbdbe06"
```

```
61 bacteria_general_results = "5f315d7c3d418525e0385fa3"
62
63 with open("bacteria_struct_update_key_collection.json", "r")
      as json_file:
64     bacteria_struct_update_key_collection = json.load(
          json_file)
65 for object_id, key, new_key in
      bacteria_struct_update_key_collection:
66     bacteria_struct.update_key_collection(object_id, key,
          new_key)
67
68
69 bacteria_struct.insert_key_collection(bacteria_main_tree, "
      OrganSampled")
70 bacteria_struct.insert_key_collection(bacteria_main_tree, "
      NSamplesIsolated")
71 bacteria_struct.insert_key_collection(bacteria_main_tree, "
      CountryOfOrigin")
72 bacteria_struct.insert_key_collection(bacteria_main_tree, "
      PackStatusType")
73 bacteria_struct.insert_key_collection(bacteria_main_tree, "
      Distribution")
74 bacteria_struct.insert_key_collection(bacteria_main_tree, "
      PrimaryProduction")
75 bacteria_struct.insert_key_collection(bacteria_main_tree, "
      FoodSubClass")
76
```

```
77 bacteria_struct.update_nested_key_value_collection(
       bacteria_general_results, "NSamples.label", "Total␣units␣
       tested")
78
79
80 with open("bacteria_struct_update_nested_key_collection.json"
       , "r") as json_file:
81     bacteria_struct_update_nested_key_collection = json.load(
           json_file)
82 for object_id, old_key_path, new_key_path in
       bacteria_struct_update_nested_key_collection:
83     bacteria_struct.update_nested_key_collection(object_id,
           old_key_path, new_key_path)
84
85
86 with open("
       bacteria_struct_insert_insert_nested_key_collection.json",
        "r") as json_file:
87     bacteria_struct_insert_insert_nested_key_collection =
           json.load(json_file)
88 for object_id, key_path, new_key_value in
       bacteria_struct_insert_insert_nested_key_collection:
89     bacteria_struct.insert_nested_key_collection(object_id,
           key_path, new_key_value)
90
91 """
92             parasite_struct
93 """
```

```
94 parasite_struct = SingleRecordUpdater(uri, database, "
       parasite_struct")
95
96 parasite_main_tree = '5f8e30e1babfbb3f18364f60'
97 parasite_general_results = '5f730806e78a2828f8b3a2df'
98 parasites = '5f7307cee78a2828f8b3a2de'
99
100 parasite_struct.update_key_collection(parasite_main_tree, "
       FoodOrigin", "CountryOfSampling")
101 parasite_struct.update_key_collection(parasite_main_tree, "
       Stage", "SamplingStage")
102 parasite_struct.update_key_collection(
       parasite_general_results, "NSamples", "TotalUnitsTested")
103 parasite_struct.update_key_collection(parasite_main_tree, "
       NSamples", "TotalUnitsTested")
104
105 parasite_struct.insert_key_collection(parasite_main_tree, "
       CountryOfOrigin")
106 parasite_struct.insert_key_collection(parasite_main_tree, "
       PackStatusType")
107 parasite_struct.insert_key_collection(parasite_main_tree, "
       Distribution")
108 parasite_struct.insert_key_collection(parasite_main_tree, "
       PrimaryProduction")
109 parasite_struct.insert_key_collection(parasite_main_tree, "
       FoodSubClass")
110
```

```
111 parasite_struct.update_nested_key_value_collection(
    parasite_general_results, "NSamples.label", "Total␣units␣
    tested")
112
113 parasite_struct.update_nested_key_collection(parasites, "
    Cryptosporidium.EssayDetType.Reference␣methods.ISO␣18744␣
    (2016)", "Cryptosporidium.EssayDetType.Reference␣methods.
    ISO␣18744:2016␣(Cryptosporidium␣&␣Giardia)")
114 parasite_struct.update_nested_key_collection(parasites, "
    Cryptosporidium.EssayDetType.Reference␣methods.ISO␣15553␣
    (2006)", "Cryptosporidium.EssayDetType.Reference␣methods.
    ISO␣15553:2006␣(Cryptosporidium␣&␣Giardia)")
115 parasite_struct.update_nested_key_collection(parasites, "
    Giardia.EssayDetType.Reference␣methods.ISO␣18744:2016␣(IMS
    +IFM)␣(produce):␣centrifugation+IMS-CG+IFA", "Giardia.
    EssayDetType.Reference␣methods.ISO␣18744:2016␣(
    Cryptosporidium␣&␣Giardia)")
116 parasite_struct.update_nested_key_collection(parasites, "
    Giardia.EssayDetType.Reference␣methods.ISO␣15553:2006␣(
    water)", "Giardia.EssayDetType.Reference␣methods.ISO␣
    15553:2006␣(Cryptosporidium␣&␣Giardia)")
117
118 parasite_struct.insert_nested_key_collection(parasites, "
    Cryptosporidium.EssayEnumType.Molecular.ISO␣18744:2016␣(
    Cryptosporidium␣&␣Giardia)")
119 parasite_struct.insert_nested_key_collection(parasites, "
    Cryptosporidium.EssayEnumType.Molecular.ISO␣15553:2006␣(
    Cryptosporidium␣&␣Giardia)")
```

```
120 parasite_struct.insert_nested_key_collection(parasites, "
    Cryptosporidium.EssayEnumType.Molecular.Other")
121 parasite_struct.insert_nested_key_collection(parasites, "
    Giardia.EssayEnumType.Molecular.ISO␣18744:2016␣(
    Cryptosporidium␣&␣Giardia)")
122 parasite_struct.insert_nested_key_collection(parasites, "
    Giardia.EssayEnumType.Molecular.ISO␣15553:2006␣(
    Cryptosporidium␣&␣Giardia)")
123 parasite_struct.insert_nested_key_collection(parasites, "
    Giardia.EssayEnumType.Molecular.Other")
124
125
126 """
127             virus_struct
128 """
129 virus_struct = SingleRecordUpdater(uri, database, "
    virus_struct")
130
131 virus_main_tree = '5ffc3f25c32f7210dcddb65a'
132 virus_general_results = '5ffc2b80c32f7210dcddb656'
133 virus_tree = "5ffc00b7c32f7210dcddb653"
134
135 virus_struct.update_key_collection(virus_main_tree, "
    FoodOrigin", "CountryOfSampling")
136 virus_struct.update_key_collection(virus_main_tree, "Stage",
    "SamplingStage")
137 virus_struct.update_key_collection(virus_general_results, "
    NSamples", "TotalUnitsTested")
```

```
138 virus_struct.update_key_collection(virus_main_tree, "NSamples
    ", "TotalUnitsTested")
139
140 virus_struct.insert_key_collection(virus_main_tree, "
    CountryOfOrigin")
141 virus_struct.insert_key_collection(virus_main_tree, "
    PackStatusType")
142 virus_struct.insert_key_collection(virus_main_tree, "
    Distribution")
143 virus_struct.insert_key_collection(virus_main_tree, "
    PrimaryProduction")
144 virus_struct.insert_key_collection(virus_main_tree, "
    FoodSubClass")
145
146 virus_struct.update_nested_key_value_collection(
    virus_general_results, "NSamples.label", "Total␣units␣
    tested")
147
148 virus_struct.update_multiple_keys("Hepatitis␣A␣(HAV)", "
    Hepatovirus␣A")
149 virus_struct.update_multiple_keys("Hepatitis␣E␣(HEV)", "
    Orthohepevirus␣A")
150
151 with open("virus_struct_update_nested_key_collection.json", "
    r") as json_file:
152     virus_struct_update_nested_key_collection = json.load(
        json_file)
153 for object_id, old_key_path, new_key_path in
    virus_struct_update_nested_key_collection:
```

```
154     virus_struct.update_nested_key_collection(object_id,
            old_key_path, new_key_path)
155
156 virus_struct.insert_nested_key_collection(virus_tree, "
    Hepatovirus␣A.EssayEnumType.Reference␣PCR␣methods.ISO␣
    15216-1:2017/Amd␣1:2021␣(Hepatitis␣A␣&␣norovirus)")
157 virus_struct.insert_nested_key_collection(virus_tree, "
    Norovirus.EssayEnumType.Reference␣PCR␣methods.ISO␣
    15216-1:2017/Amd␣1:2021␣(Hepatitis␣A␣&␣norovirus)" )
158
159 """
160             food_struct
161 """
162
163 food_struct = SingleRecordUpdater(uri, database, "food_struct
    ")
164 food_characteristics_tree = '5f315e123d418525e0385fa6'
165
166 food_struct.remove_key_collection(food_characteristics_tree,
    "Stage")
167 food_struct.remove_key_collection(food_characteristics_tree,
    "PackStatus")
```

Listing B.2: MultiRecordUpdater class

```
1 import pymongo
2 from bson import ObjectId
3
4 class MultiRecordUpdater:
5     def __init__(self, uri, database, collection_name):
```

```
        self.client = pymongo.MongoClient(uri)
        self.database = database
        self.collection = collection_name
        self.db_collection = self.client[self.database][self.
            collection]

    def update_value_collection(self, key, value, new_value):
        if value == new_value:
            return

        self.db_collection.update_many(
            {key: value},
            {"$set": {key: new_value}}
        )

    def update_key_collection(self, key, new_key):
        if key == new_key:
            return

        self.db_collection.update_many(
            {},
            {"$rename": {key: new_key}}
        )
```

Listing B.3: SingleRecordUpdater class

```
import pymongo
from bson import ObjectId

import pymongo
```

```
5  from bson import ObjectId
6
7  class SingleRecordUpdater:
8      def __init__(self, uri, database, collection):
9          self.client = pymongo.MongoClient(uri)
10         self.database = database
11         self.collection = collection
12         self.db_collection = self.client[database][self.
               collection]
13
14     def update_key_collection(self, object_id, key, new_key):
15         object_id = ObjectId(object_id)
16
17         self.db_collection.update_one(
18             {"_id": object_id},
19             {"$rename": {key: new_key}}
20         )
21
22     def insert_key_collection(self, object_id, key, key_value
           =None):
23         object_id = ObjectId(object_id)
24
25         self.db_collection.update_one(
26             {"_id": object_id},
27             {"$set": {key: key_value}}
28         )
29
30     def remove_key_collection(self, object_id, key):
31         object_id = ObjectId(object_id)
```

```
32
33          self.db_collection.update_one(
34              {"_id": object_id},
35              {"$unset": {key: ""}}
36          )
37
38      def update_multiple_keys(self, key, new_key):
39
40          self.db_collection.update_many(
41              {},
42              {"$rename": {key: new_key}}
43          )
44
45      def update_nested_key_value_collection(self, object_id,
           key_path, new_value):
46          object_id = ObjectId(object_id)
47
48          set_statement = {f"{key_path}": new_value}
49          self.db_collection.update_one(
50              {"_id": object_id},
51              {"$set": set_statement}
52          )
53
54      def update_nested_key_collection(self, object_id,
           old_key_path, new_key_path):
55          object_id = ObjectId(object_id)
56
57          self.db_collection.update_one(
58              {"_id": object_id},
```

```
59              {"$rename": {old_key_path: new_key_path}}
60          )
61
62      def insert_nested_key_collection(self, object_id,
           key_path, new_key_value=None):
63          object_id = ObjectId(object_id)
64
65          self.db_collection.update_one(
66              {"_id": object_id},
67              {"$set": {key_path: new_key_value}}
68          )
```