



University of Camerino

SCHOOL OF SCIENCE AND TECHNOLOGIES

Course Master Degree In Computer Science (LM-18)

Knowledge Engineering and Business Intelligence

Personalized Menu Recommendations in Restaurants Using Knowledge Graphs, Ontologies, and Agile BPMN Processes.

Student

Medhin Berhe Hagos (MAT 127546)

Professors

**Prof. Knut Hinkelmann
Prof. Emanuele Laurenzi**

A.A. 2024/2025

Contents

1	Introduction	7
1.1	Project Objective.....	7
1.2	Menu Composition.....	7
2	Decision Model: Camunda	11
2.1	Decision Model.....	11
2.2	Knowledge Sources	12
2.3	Input	12
2.4	Decision Tables.....	13
2.5	DMN Simulation.....	14
3	PROLOG	15
3.1	Prolog rules	16
3.2	Prolog Simulation	18
4	Ontology Engineering	20
4.1	Ontology	20
4.1.1	Classes.....	22
4.2	SWRL Rules.....	26
4.3	SHACL Shapes.....	31
4.4	SPARQL Queries.....	38
5	Agile and Ontology-based Meta-modelling	41
5.1	AOAME	41
5.1.1	BPMN model	42
5.2	Jena Fuseki	43
5.2.1	Query in Jena Fuseki	43
6	conclusion	45

Listings

4.1	Ingredient Shape.....	31
4.2	Gluten Ingredient Shape.....	32
4.3	NoGluten Ingredient Shape	32
4.4	Lactose Ingredient Shape.....	32
4.5	NoLactose Ingredient Shape	32
4.6	VegIn gredient Shape.....	33
4.7	NoVeg Ingredient Shape	33
4.8	Meal Shape.....	33
4.9	Gluten Meal Shape	34
4.10	NoGluten Meal Shape.....	35
4.11	Lactose Meal Shape	35
4.12	NoLactose Meal Shape	35
4.13	Veg Meal Shape	36
4.14	NoVeg Meal Shape	36
4.15	Low Calories Meal Shape	36
4.16	Medium Calories Meal Shape	37
4.17	High Calories Meal Shape	37
4.18	Menu Shape.....	37

List of Figures

2.1	Camunda model.	11
2.2	Knowledge Sources: Ingredients and Meals.	12
2.3	Client input.	12
2.4	Selected ingredients decision table.	13
2.5	Final meals decision table.	14
3.1	Example of how ingredients are described in PROLOG	15
3.2	Example of how meals are described in PROLOG	16
3.3	Rules to check if the ingredient contains lactose, gluten and if it is vegetarian.	16
3.4	Check the meals.	17
3.5	Check calories meals.	17
3.6	Check preferences predicate.	17
3.7	Return the final meals based on client preferences.	18
3.8	Prolog simulation with lactose free, gluten free and vegetarian and a low calorie level.	18
3.9	Prolog simulation with no specific preferences and an high calorie level.	19
3.10	Prolog simulation with preference for gluten-free and medium calorie level meals.	19
4.1	Structure of the classes of my model in Protégé	21
4.2	Graph of classes in Protégé	21
4.3	Lists with some of the objects of Ingredient and Meal.	22
4.4	Example of Ingredient object.	23
4.5	Ingredient sub classes.	23
4.6	Example of Meal object.	24
4.7	List of Meal subclasses.	25
4.8	Example of a Menu object.	25
4.9	Result of the vegetarian ingredient rule.	26
4.10	Result of the non-vegetarian ingredient rule.	27
4.11	Result of the vegetarian meals rule.	27
4.12	Result of the non-vegetarian meals rule.	28
4.13	Result of the low calories meals rule.	28
4.14	Result of the reverse of hasIngredients rule for the Ingredient object <i>Flour</i>	29
4.15	Result of the lactose gluten non vegetarian high calories level menu rule.	29

4.16	Result of the lactose-free gluten-free vegetarian low calories level menu rule.	30
4.17	Result of the validation of the <i>shacl</i> <i>shapes</i> file.	38
4.18	Queries prefixes.	38
4.19	Query to get all the non-vegetarian ingredients.	39
4.20	Result of the query in Figure 4.19.	39
4.21	Query that returns the meals that match the client's preferences.	40
4.22	Result of the final query if the client doesn't have any allergies, is not vegetarian but only wants low calorie meals.	40
5.1	BPMN model that describes the process of personalization of a menu in a restaurant.	42
5.2	Example of the properties for Create Personalized Menu where the client is intolerant to lactose and gluten.	42
5.3	Query to get the preferred meals only.	43
5.4	Result of the query in Figure 5.3 with the values in Figure 5.2.	44

List of Tables

1.1	Ingredients and Dietary Attributes.....	9
1.2	Meals, Ingredients and Calories Levels	10

1. Introduction

In today's rapidly evolving technological landscape, many sectors are embracing advancements to optimize their operations and stay competitive. In the restaurant industry, a common adaptation is the shift towards digital menus accessible via QR codes—a convenient solution that reduces printing costs and streamlines menu updates. However, while this offers clear benefits for restaurant owners, customers may not benefit as much, especially when using devices with small screens. Reading a large and complex menu can become cumbersome, diminishing the dining experience.

In this scenario, menu personalization becomes a crucial factor. Adapting the menu to the customers' preferences and dietary needs—such as allergies (lactose or gluten), vegetarian choices, and caloric preferences (low, medium, or high)—can significantly improve the overall customer experience and satisfaction.

1.1 Project Objective

The purpose of this project is to design and develop a sophisticated Knowledge Base aimed at generating personalized menus tailored to customers' specific dietary needs and preferences. This initiative brings together a variety of advanced tools and technologies to effectively address the complexity of the task.

To achieve this goal, various tools and technologies were used: Camunda (Chapter 2) has been utilized for modeling and managing decision-making processes, while Prolog (Chapter 3) underpins the implementation of Rule-Based Systems. Protegé (Chapter 4) has played a key role in Ontology Engineering, ensuring a robust and flexible conceptual framework for the Knowledge Base. Additionally, AOAME and Jena Fuseki 5) provide crucial support for graphical data representation and efficient query execution.

Menu Composition

This section offers a comprehensive overview of the menu structure employed in my system, which includes a total of 44 distinct ingredients and 36 carefully curated dishes. Each ingredient is characterized by several attributes to ensure a detailed and accurate representation within the Knowledge Base. Specifically, every ingredient is defined by its name, lactose content, gluten content, whether it is suitable for vegetarians (based on the presence or absence of meat), and its calorie count. These attributes allow for precise filtering and selection to accommodate diverse dietary needs and preferences (see Table 1.1).

On the other hand, each dish is described by its name and the list of ingredients that compose it. The choice of ingredients for each dish is guided by their individual calorie counts, ensuring that the overall nutritional value of the dish aligns with specific dietary considerations. This structured approach not only facilitates the creation of balanced dishes but also ensures consistency and adaptability when generating personalized menus.

By structuring the menu in this way, the system is equipped to handle a wide variety of dietary requirements, making it both flexible and comprehensive in its ability to cater to users' needs.

Table 1.1: Ingredients and Dietary Attributes

Ingredients	Lactose Intolerant	Gluten Free	Vegetarian	Vegan	Kcal/100gr
Coffee	yes	Yes	yes	yes	0
Water	yes	Yes	yes	yes	0
Sparkling Water	yes	Yes	yes	yes	0
Sugar	yes	Yes	yes	yes	387
Salt	yes	Yes	yes	yes	0
Rosemary	yes	Yes	yes	yes	131
Oil	yes	Yes	yes	yes	884
Flour	yes	No	yes	yes	364
Pasta	yes	No	yes	yes	131
Bread	yes	No	yes	yes	265
Lemon	yes	Yes	yes	yes	29
Onion	yes	Yes	yes	yes	40
Lettuce	yes	Yes	yes	yes	15
Zucchini	yes	Yes	yes	yes	17
Eggplant	yes	Yes	yes	yes	25
Peas	yes	Yes	yes	yes	81
Potato	yes	Yes	yes	yes	77
Carrot	yes	Yes	yes	yes	40
Tomato	yes	Yes	yes	yes	18
Tomato Sauce	yes	Yes	yes	yes	105
Champignon	yes	Yes	yes	yes	22
Truffle	yes	Yes	yes	yes	284
Avocado	yes	Yes	yes	yes	231
Ketchup	yes	Yes	yes	yes	112
Mayonnaise	yes	Yes	yes	no	680
Ricotta	no	Yes	yes	no	174
Pecorino	no	Yes	yes	no	383
Parmesan	no	Yes	yes	no	431
Mozzarella	no	Yes	yes	no	280
Mascarpone	no	Yes	yes	no	450
Salmon	yes	Yes	yes	no	208
Seabass	yes	Yes	yes	no	206
Duck	yes	Yes	no	no	160
Clams	yes	Yes	yes	no	148
Beef	yes	Yes	no	no	143
Bacon	yes	Yes	no	no	541
Sausage	yes	Yes	no	no	346
Ham	yes	Yes	no	no	145
Egg	yes	Yes	yes	no	155
Rice	yes	Yes	yes	yes	130
Strawberry	yes	Yes	yes	yes	33
Peach	yes	Yes	yes	yes	25
Apple	yes	Yes	yes	yes	52
Ladyfingers	yes	No	yes	yes	50

Table 1.2: Meals, Ingredients and Calories Levels

Meal	Ingredients	Kcal Value
Tartare	Salmone, Sale, Avocado, Limone, Pomodoro	Alto
Arancini al Ragù	Pane, Farina, Riso, Sale, Acqua, Mozzarella, Cipolla, Manzo, Olio, Sugo di Pomodoro	Alto
Arancini al Prosciutto	Pane, Farina, Riso, Sale, Acqua, Prosciutto, Mozzarella	Medio
Arancini Vegetariani	Pane, Farina, Riso, Sale, Acqua, Cipolla, Olio, Sugo di Pomodoro, Mozzarella	Medio
Tagliere	Prosciutto, Mozzarella, Pecorino, Ricotta, Parmesan	Alto
Insalata Russa	Patata, Piselli, Carota, Uovo, Olio, Sale, Maionese	Alto
Carbonara	Pasta, Uovo, Pecorino, Bacon	Alto
Pasta Salmone e Pomodoro	Pasta, Salmone, Sugo di Pomodoro, Sale, Olio	Basso
Pasta alle Vongole	Pasta, Vongole, Olio, Sale	Basso
Gnocchi al Sugo di Papera	Farina, Patata, Uovo, Sale, Anatra, Cipolla, Carota, Sugo di Pomodoro, Parmesan, Olio	Alto
Risotto alle Vongole	Riso, Vongole, Olio, Sale	Basso
Risotto Salsiccia e Funghi	Riso, Salsiccia, Champignon, Cipolla, Olio, Sale	Medio
Ravioli al Ragù	Pecorino, Ricotta, Sale, Farina, Uovo, Cipolla, Sugo di Pomodoro, Manzo, Olio	Alto
Ravioli Vegetariani	Pecorino, Ricotta, Sale, Farina, Uovo, Cipolla, Sugo di Pomodoro, Olio	Medio
Parmigiana di Melanzane	Melanzana, Sugo di Pomodoro, Mozzarella, Parmesan, Cipolla, Olio, Sale	Alto
Arrosto Misto di Carne	Manzo, Salsiccia, Limone, Sale, Olio, Rosmarino	Medio
Grigliata Mista di Carne	Manzo, Salsiccia, Limone, Sale, Olio, Rosmarino	Basso
Arrosto Misto di Pesce	Salmone, Sale, Olio, Branzino, Rosmarino, Limone	Medio
Grigliata Mista di Pesce	Salmone, Sale, Olio, Branzino, Rosmarino, Limone	Basso
Scaloppine ai Funghi	Manzo, Champignon, Farina, Olio, Sale, Rosmarino	Basso
Melanzane Ripiene	Melanzana, Maiale, Sugo di Pomodoro, Cipolla, Pane, Olio, Sale, Parmesan	Alto
Insalata	Lattuga, Pomodoro, Carota, Olio, Sale	Basso
Patate al Forno	Patata, Olio, Sale, Rosmarino	Medio
Patatine Fritte	Patata, Sale, Olio, Maionese, Ketchup	Alto
Verdure Grigliate	Melanzana, Pomodoro, Olio, Sale, Zucchini, Rosmarino	Basso
Pizza Margherita	Sale, Acqua, Farina, Olio, Mozzarella, Sugo di Pomodoro	Medio
Pizza Salmone	Sale, Acqua, Farina, Olio, Mozzarella, Salmone	Medio
Pizza alla Boscaiola	Sale, Acqua, Farina, Olio, Mozzarella, Champignon, Salsiccia	Alto
Pizza alla Norcina	Sale, Acqua, Farina, Olio, Mozzarella, Champignon, Tartufo, Salsiccia	Alto
Pizza Vegetariana	Sale, Acqua, Farina, Olio, Mozzarella, Melanzana, Cipolla, Zucchini, Carota, Pomodoro, Champignon	Medio
Pizza con Patate e Bacon	Sale, Acqua, Farina, Olio, Mozzarella, Sugo di Pomodoro, Patata, Bacon	Alto
Pizza con Prosciutto	Sale, Acqua, Farina, Olio, Mozzarella, Prosciutto, Sugo di Pomodoro	Alto
Pizza Zucchine e Salsiccia	Sale, Acqua, Farina, Olio, Mozzarella, Zucchini, Salsiccia	Alto
Pizza Melanzana e Prosciutto	Sale, Acqua, Farina, Olio, Mozzarella, Melanzana, Prosciutto, Sugo di Pomodoro	Alto
Tiramisù	Caffè, Mascarpone, Savoiardi	Alto
Macedonia	Zucchero, Limone, Fragola, Pesca, Mela	Basso

2. Decision Model: Camunda

Camunda allows for modeling business decisions using the DMN language, a standard that provides a visual and formal representation of decision rules. This approach helps to define decision logic clearly and understandably, enabling companies to create models that reflect their decision-making policies and procedures.

Camunda uses decision diagrams to represent decisions and decision rules. The diagrams can include decision tables, which display rules and criteria in a tabular format, making it easier to understand and manage complex decisions.

Once modeled, decisions can be executed directly within the Camunda platform. Camunda integrates decision rules into business processes and automates them, enhancing the efficiency and consistency of decisions.

2.1 Decision Model

The image 2.1 shows My decision model, which takes as input the list of available ingredients and the list of all dishes included in the menu. Additionally, the model incorporates the customer's dietary preferences, such as lactose intolerance, gluten intolerance, vegetarian choice, and the desired calorie level.

As can be seen, the model has been simplified compared to the previous version. Following the feedback received, i have modified the definitions of the rules within the decision tables to ensure that the execution is less script-like and more aligned with the decision-making paradigm. The rules in the decision tables are explained in the following sections.

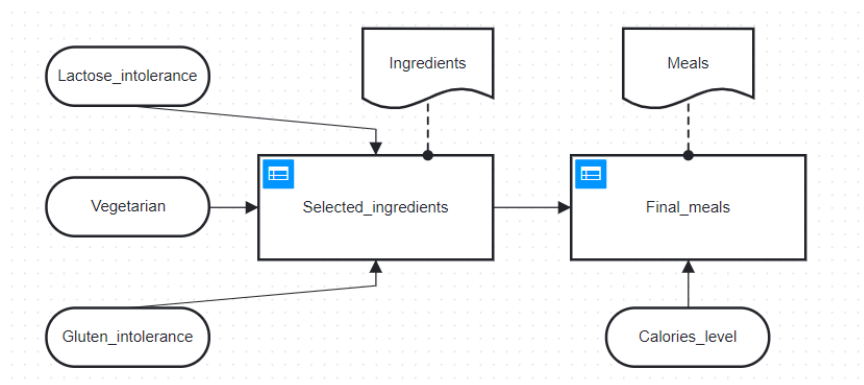


Figure 2.1: Camunda model.

2.2 Knowledge Sources

Knowledge Sources represent the fundamental sources of knowledge that provide the data and information necessary to support the decision-making process. They are considered fixed resources that offer a knowledge base on which to apply decision rules and determine the most appropriate solution based on specific inputs.

In my diagram, there are two Knowledge Sources: Ingredients and Meals [image 2.2]. Ingredients represents the list of available ingredients in the restaurant, describing the ingredients at a theoretical level, while Meals represents the list of all the dishes available in the restaurant.



Figure 2.2: Knowledge Sources: Ingredients and Meals.

2.3 Input

Input Data represent the data provided to the system as input for the decision-making process. These data are typically variables that the user or another system provides during the execution of the process. In DMN, Input Data are used by the Decision Tables to calculate or determine the output.

In my diagram, there are three Input Data used to allow the customer to input possible intolerances and dietary preferences: Lactose_intolerance, Gluten_intolerance, Vegetarian and Calories_level [image 2.3].

These Input Data are of a boolean type. Therefore, if a customer is lactose intolerant, they must enter true, otherwise false. The same logic applies to gluten intolerance and vegetarian preferences. Calories_level allows the customer to specify the preferred calorie level. If not specified, all dishes with any available calorie levels will be considered.



Figure 2.3: Client input.

2.4 Decision Tables

The decision tables used in the model are described in detail below:

- **Selected_ingredients:** It aims to identify the ingredients that should not be present in the dishes, based on the customer's preferences. Some rules use the symbol - in the conditions, indicating that the specific condition is irrelevant for that rule. For example, a rule with - in all conditions means that the corresponding ingredient is selected regardless of lactose intolerance, gluten intolerance, or vegetarian preferences, as it can be consumed by a lactose-intolerant customer, a gluten-intolerant customer, or a vegetarian. Conversely, if an ingredient is marked as false for one of the inputs, that ingredient will be excluded, and dishes containing it will not be available to the customer.

The Collect hit policy indicates that if multiple rules are satisfied simultaneously, the corresponding results will be aggregated into a list. In the context of Selected_ingredients, this means that various ingredients can be selected based on the specified conditions.

Selected_ingredients		Hit policy: Collect		
	When	And	And	Then
	Lactose_intolerance	Gluten_intolerance	Vegetarian	Selected_ingredients
	boolean	boolean	boolean	"Coffee","Water","Sparkling Wat...
27	-	-	-	"Rice"
28	-	-	-	"Strawberry"
29	-	-	-	"Peach"
30	-	-	-	"Apple"
31	false	-	-	"Ricotta"
32	false	-	-	"Pecorino"
33	false	-	-	"Parmesan"
34	false	-	-	"Mozzarella"
35	false	-	-	"Mascarpone"
36	-	false	-	"Flour"
37	-	false	-	"Pasta"
38	-	false	-	"Bread"
39	-	false	-	"Ladyfingers"
40	-	-	false	"Beef"
41	-	-	false	"Bacon"
42	-	-	false	"Sausage"
43	-	-	false	"Ham"

Figure 2.4: Selected ingredients decision table.

- **Final meals:** The objective of this decision table is to determine and recommend a specific dish based on the selected ingredients and the desired calorie level of the customer. The decision table takes as input a list of selected ingredients (Selected_ingredients) and a calorie level, which can be "low," "medium," or "high," and outputs the name of a specific dish. In other words, the model allows for recommending the most suitable dish based on the available ingredients and the chosen calorie level. The Selected_ingredients condition specifies that a dish can only be selected if all the listed ingredients are present in the Selected_ingredients list, in order to exclude dishes that may contain ingredients the customer wants to avoid or cannot eat. The Collect hit policy allows for gathering all the dishes that the customer can eat.

Final_meals		Hit policy: Collect
When	And	Then
Selected_ingredients	Calories_level	output
4 list contains(Selected_ingredients, "Bread") and list contains(Selected_ingredients, "Flour") and list contains(Selected_ingredients, "Rice") and list contains(Selected_ingredients, "Salt") and list contains(Selected_ingredients, "Water") and list contains(Selected_ingredients, "Onion") and list contains(Selected_ingredients, "Mozzarella") and list contains(Selected_ingredients, "Beef") and list contains(Selected_ingredients, "Oil") and list contains(Selected_ingredients, "Tomato sauce")	"" "medium"	"Arancini al ragù" "Arancini vegetariani"

Figure 2.5: Final meals decision table.

2.5 DMN Simulation

Unlike the previous project, in this case, it is not possible to obtain the simulation results as Camunda no longer provides the software for simulating DMN diagrams. Below are the hypothetical results of the simulation.

Scenario 1

Client preferences: lactose, gluten, no vegetarian and medium calories level.

In this case, all ingredients are considered, and then in Final meals, dishes are associated with these ingredients that have a medium or low calorie level. Finally, a list of all available dishes is returned.

Scenario 2

Client preferences: lactose free, gluten free, vegetarian and high calories level.

In this case, the customer is lactose intolerant, gluten intolerant, and wants only vegetarian ingredients. Additionally, they have specified a high calorie level. Selected_ingredients will return only ingredients that meet these preferences, and Final_meals will return all dishes that contain only the ingredients received from Selected_ingredients. Since the calorie level is high, the customer can also eat dishes with a medium or low calorie level.

3. PROLOG

Prolog is a declarative programming language based on the logical paradigm. Users define rules and facts, and the Prolog system uses an inference engine to resolve queries. I have implemented the model defined in the DMN file using Prolog.

My implementation in Prolog is based on the DMN diagram created with Camunda. First, the ingredients and dishes were described. Then, the rules defining the behavior for creating the personalized menu were implemented.

The image 3.1 shows how the ingredients are described. Each ingredient is represented using the ingredient predicate, which specifies the name of the ingredient followed by its characteristics. Each ingredient is described by three key attributes: whether it is lactose-free, gluten-free, and whether it is vegetarian. For example, the ingredient salmon is listed as lactose_free, gluten_free, and vegetarian, while flour is listed as lactose_free, but contains gluten and is vegetarian. This structured list allows the program to later associate these ingredients with specific dishes, and facilitate the creation of personalized menus based on customer preferences.

```
1 /* Ingredients */
2 ingredient(salmon,lactose_free,gluten_free,vegetarian).
3 ingredient(coffee,lactose_free,gluten_free,vegetarian).
4 ingredient(water,lactose_free,gluten_free,vegetarian).
5 ingredient(sparkling_water,lactose_free,gluten_free,vegetarian).
6 ingredient(sugar,lactose_free,gluten_free,vegetarian).
7 ingredient(salt,lactose_free,gluten_free,vegetarian).
8 ingredient(rosemary,lactose_free,gluten_free,vegetarian).
9 ingredient(oil,lactose_free,gluten_free,vegetarian).
10 ingredient(flour,lactose_free,gluten,vegetarian).
11 ingredient(pasta,lactose_free,gluten,vegetarian).
12 ingredient(bread,lactose_free,gluten,vegetarian).
13 ingredient(lemon,lactose_free,gluten_free,vegetarian).
14 ingredient(onion,lactose_free,gluten_free,vegetarian).
15 ingredient(lettuce,lactose_free,gluten_free,vegetarian).
```

Figure 3.1: Example of how ingredients are described in PROLOG

While the image 3.2 shows how the dishes are described. Each meal is represented using the meal predicate, which includes the name of the meal, followed by a list of ingredients that make up the dish, and its corresponding caloric value (categorized as low, medium, or high). For example, the meal Tartare consists of the ingredients salt, salmon, avocado, lemon, and tomato and is classified as high in calories. Similarly, Arancini al ragù contains ingredients such as bread, flour, rice, salt, water, mozzarella, onion, beef, oil, and tomato sauce, and is classified as high in calories. Each meal's list of ingredients allows the program to determine which items should be included based on the customer's dietary preferences, facilitating the creation of a personalized menu.

```

47 /* Meals*/
48 meal("Tartare", [salt, salmon, avocado, lemon, tomato], high).
49 meal("Arancini_al_ragù", [bread, flour, rice, salt, water, mozzarella, onion, beef, oil, tomato_sauce], high).
50 meal("Arancini_vegetariani", [bread, flour, rice, salt, water, mozzarella, onion, oil, tomato_sauce], medium).
51 meal("Arancini_al_prosciutto", [bread, flour, rice, salt, water, mozzarella, ham], medium).
52 meal("Tagliere", [ham, mozzarella, pecorino, parmesan, ricotta], high).
53 meal("Insalata_russa", [potato, peas, carrot, mayonnaise, egg, oil, salt], high).
54 meal("Carbonara", [pasta, egg, pecorino, bacon], high).
55 meal("Pasta_salmone_pomodoro", [pasta, salmon, tomato_sauce, salt, oil], low).
56 meal("Pasta_alle_vongole", [pasta, clams, oil, salt], low).
57 meal("Gnocchi_al_sugo_di_papera", [flour, potato, egg, salt, duck, onion, carrot, tomato_sauce, parmesan, oil], high).
58 meal("Risotto_alle_vongole", [rice, clams, oil, salt], low).
59 meal("Risotto_salsiccia_funghi", [rice, sausage, champignon, onion, oil, salt], medium).
60 meal("Ravioli_al_ragù", [pecorino, ricotta, salt, flour, egg, onion, tomato_sauce, beef, oil], high).
61 meal("Ravioli_vegetariani", [pecorino, ricotta, salt, flour, egg, onion, tomato_sauce, oil], medium).
62 meal("Parmigiana", [eggplant, tomato_sauce, mozzarella, parmesan, onion, oil, salt], high).
63 meal("Arrosto_di_carne", [beef, sausage, lemon, salt, oil, rosemary], medium).
64 meal("Grigliata_di_carne", [beef, sausage, lemon, salt, oil, rosemary], low).
65 meal("Arrosto_di_pesce", [seabass, salmon, lemon, salt, oil, rosemary], medium).
66 meal("Grigliata_di_pesce", [seabass, salmon, lemon, salt, oil, rosemary], low).
67 meal("Scaloppine_ai_funghi", [beef, champignon, flour, oil, salt, rosemary], low).
68 meal("Melanzane_ripiene", [eggplant, beef, tomato_sauce, onion, bread, oil, salt, parmesan], high).
69 meal("Insalata", [lettuce, tomato, carrot, oil, salt], low).

```

Figure 3.2: Example of how meals are described in PROLOG

3.1 Prolog rules

The image 3.3 shows the three predicates that check specific dietary characteristics of ingredients.

- *is_lactose_free(X)*: this predicate checks if an ingredient X is lactose-free. It does so by querying the ingredient facts where the second argument is `lactose_free`. If the ingredient is defined with this characteristic, the predicate will return true.
- *is_gluten_free(X)*: this predicate verifies if an ingredient X is gluten-free. It checks the ingredient facts where the third argument is `gluten_free`. If the ingredient is marked as gluten free, the predicate returns true.
- *is_vegetarian(X)*: this predicate determines whether an ingredient X is vegetarian. It looks for the vegetarian attribute in the fourth argument of the ingredient facts. If the ingredient is defined as vegetarian, the predicate will return true.

```

87 /*Allergies + Vegetarian*/
88 is_lactose_free(X) :- ingredient(X,lactose_free,_,_)
89 is_gluten_free(X) :- ingredient(X,_,gluten_free,_).
90 is_vegetarian(X) :- ingredient(X,_,_,vegetarian).

```

Figure 3.3: Rules to check if the ingredient contains lactose, gluten and if it is vegetarian.

The image 3.4 shows three rules used to check if dishes contain allergenic ingredients and if they are vegetarian. The rule `meal_lactose_free(Meal)` checks if a meal is lactose-free by verifying that all ingredients in the meal are free from lactose. The rule `meal_gluten_free(Meal)` checks if a meal is gluten-free by ensuring that each ingredient in the meal's list does not contain gluten. The rule `meal_vegetarian(Meal)` determines if a meal is vegetarian by verifying that all ingredients in the meal are vegetarian.


```

96 meal_lactose_free(Meal) :- meal(Meal, Ingredients, _),
97     forall(member(Ingredient, Ingredients), is_lactose_free(Ingredient)).
98
99 meal_gluten_free(Meal) :-
00     meal(Meal, Ingredients, _),
01     forall(member(Ingredient, Ingredients), is_gluten_free(Ingredient)).
02
03 meal_vegetarian(Meal) :-
04     meal(Meal, Ingredients, _),
05     forall(member(Ingredient, Ingredients), is_vegetarian(Ingredient)).

```

Figure 3.4: Check the meals.

The image 3.5 shows the rule `meal_by_calories` (Calorie Level, Meal) finds all meals (Meal) that meet the specified calorie criterion (Calorie Level). It uses the numeric values associated with the calorie levels to compare the calorie level of each meal with the required level. If the calorie level of the meal is less than or equal to the required level, the meal is selected.

```

111 % Calories order
112 calorie_order(high, 3).
113 calorie_order(medium, 2).
114 calorie_order(low, 1).
115
116 % Find dishes that meet the calorie criterion
117 meal_by_calories(CalorieLevel, Meal) :-
118     calorie_order(CalorieLevel, Level),
119     meal(Meal, _, Calories),
120     calorie_order(Calories, MealLevel),
121     MealLevel =< Level.

```

Figure 3.5: Check calories meals.

The image 3.6 shows the rules `check_preferences` (Meal, Preferences). The predicate is designed to evaluate whether a given meal satisfies the dietary preferences specified by the client. It takes two arguments: the first is the meal being analyzed, and the second is a list of preferences that may include `lactose_free`, `gluten_free`, or `vegetarian`. For each preference present in the list, the predicate performs a specific check. If `lactose_free` is included, the predicate calls `meal_lactose_free/1` to ensure that the meal does not contain any lactose. Similarly, if `gluten_free` is listed, it invokes `meal_gluten_free/1` to verify the absence of gluten. Finally, if `vegetarian` is specified, the predicate checks the meal with `meal_vegetarian/1` to confirm its suitability for vegetarians. If a particular preference is not provided, the corresponding check is skipped, and the condition evaluates to true.

```

check_preferences(Meal, Preferences) :-
    (member(lactose_free, Preferences) -> meal_lactose_free(Meal) ; true),
    (member(gluten_free, Preferences) -> meal_gluten_free(Meal) ; true),
    (member(vegetarian, Preferences) -> meal_vegetarian(Meal) ; true).

```

Figure 3.6: Check preferences predicate.

The predicat `find_meals` (Preferences, Calorie Level, Meals) (image 3.7) is designed to retrieve a list of meals that satisfy the client's dietary preferences and calorie level requirements. It takes three arguments: the first is a list of preferences (e.g., `lactose_free`, `gluten_free`, `vegetarian`), the second specifies the desired calorie level (e.g., `low`, `medium`, `high`), and the third is the resulting list of meals that meet the criteria. The predicate uses `findall/3` to construct this list. For each meal, it ensures the following conditions are satisfied: the meal exists in the database (`meal/3`), the meal matches the user's preferences as verified by `check_preferences/2`, and the meal aligns with the specified calorie level as determined by `meal_by_calories/2`. Meals that fulfill all these conditions are added to the result list, making this predicate an essential tool for generating personalized menu recommendations tailored to individual dietary and caloric needs.

```
find_meals(Preferences, CalorieLevel, Meals) :-
    findall(
        Meal,
        (
            meal(Meal, _, _),
            check_preferences(Meal, Preferences),
            meal_by_calories(CalorieLevel, Meal)
        ),
        Meals
    ).
```

Figure 3.7: Return the final meals based on client preferences.

3.2 Prolog Simulation

The simulation in Figure 3.8 considers a customer who can't eat foods that contain lactose, gluten, and who is vegetarian, and include meals with low calories. The returned list contains the meals that meet these preferences.

```
Meals = ["Risotto_alle_vongole", "Grigliata_di_pesce", "Insalata", "Verdure_grigliate", "Macedonia"]
```

```
?- find_meals([lactose_free, gluten_free, vegetarian], low, Meals)
```

Figure 3.8: Prolog simulation with lactose free, gluten free and vegetarian and a low calorie level.

The simulation in Figure 3.9 considers a customer who can eat foods that contain lactose, gluten, may not be vegetarian, and include meals with high calories. The returned list contains the meals that meet these preferences.

```

Meals =
["Tartare", "Arancini_al_ragù", "Arancini_vegetariani", "Arancini_al_prosciutto", "Tagliere", "Insalata_russa",
"Carbonara", "Pasta_salmone_pomodoro", "Pasta_alle_vongole", "Gnocchi_al_sugo_di_paperera",
"Risotto_alle_vongole", "Risotto_salsiccia_funghi", "Ravioli_al_ragù", "Ravioli_vegetariani", "Parmigiana",
"Arrosto_di_carne", "Grigliata_di_carne", "Arrosto_di_pesce", "Grigliata_di_pesce", "Scaloppine_ai_funghi",
"Melanzane_ripiene", "Insalata", "Patate_al_forno", "Patatine_fritte", "Verdure_grigliate", "Pizza_margherita",
"Pizza_salmone", "Pizza_boscaiola", "Pizza_norcina", "Pizza_vegetariana", "Pizza_patate_e_bacon",
"Pizza_prosciutto", "Pizza_melanzane_e_prosciutto", "Pizza_zucchine_e_salsiccia", "Tiramisù", "Macedonia"]

?- find_meals([lactose, gluten, no_vegetarian], high, Meals)

```

Figure 3.9: Prolog simulation with no specific preferences and an high calorie level.

The simulation in Figure 3.10 considers a customer who can eat foods that contain lactose, can't eat gluten, may not be vegetarian, and include meals with medium calories. The returned list contains the meals that meet these preferences.

```

Meals =
["Risotto_alle_vongole", "Risotto_salsiccia_funghi", "Arrosto_di_carne", "Grigliata_di_carne", "Arrosto_di_pesce",
"Grigliata_di_pesce", "Insalata", "Patate_al_forno", "Verdure_grigliate", "Macedonia"]

?- find_meals([lactose, gluten_free, no_vegetarian], medium, Meals)

```

Figure 3.10: Prolog simulation with preference for gluten-free and medium calorie level meals.

4. Ontology Engineering

Ontology engineering is a crucial field within knowledge representation that focuses on the creation, development, and management of ontologies. An Ontology, in this context, is a formal and explicit specification of a shared conceptualization. It defines the entities within a domain, the relationships between them, and the rules governing their interactions, providing a structured framework for understanding and organizing knowledge.

Turtle Syntax

The Ontology i defined was saved in a Turtle file (TTL). Turtle (TTL) is a popular syntax for expressing RDF (Resource Description Framework) data in a human-readable and concise format. RDF is a framework used for representing information about resources on the web in a structured way, using triples consisting of a subject, predicate, and object.

Turtle enhances the readability of RDF data by allowing for the use of prefixes to abbreviate long URIs (Uniform Resource Identifiers). This makes it easier for users to write, read, and understand RDF data compared to other serialization formats like RDF/XML.

Protégé

Protégé is the software we used for ontology engineering tasks. Developed by Stanford University, Protégé is an open-source ontology editor that offers a robust and user-friendly environment for creating, editing, and managing ontologies. Protégé supports a range of ontology languages, including OWL (Web Ontology Language) and RDF (Resource Description Framework). [Mus15]

4.1 Ontology

Compared to initial submission , i have refined the structure of the classes of ontology by introducing the Menu class, as shown in Figure 4.1 and Figure 4.2. This new class was added to represent and group personalized menus tailored to the preferences specified by individual clients.

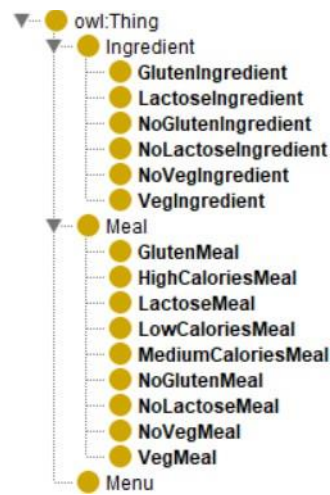


Figure 4.1: Structure of the classes of my model in Protégé.

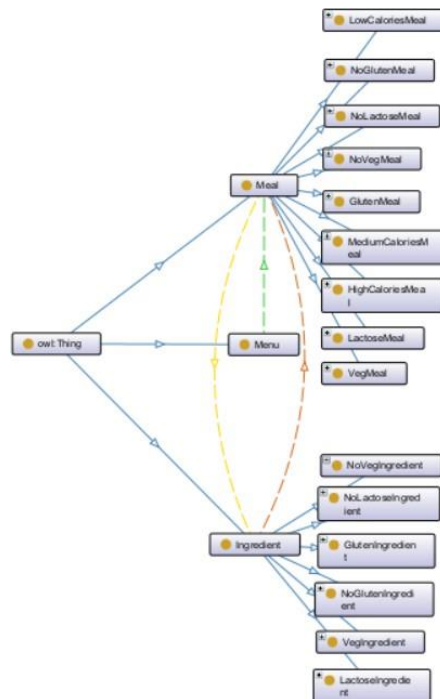


Figure 4.2: Graph of my classes in Protégé.

4.1.1 Classes

The ontology is structured around three primary classes: *Ingredient*, *Meal* and *Menu*, as illustrated in Figure 4.1. Each class plays a distinct role in representing the

key elements of the domain. The *Ingredient* class encompasses all the basic components used to prepare meals (see Table 1.1). The *Meal* class defines individual dishes, characterized by their ingredients (see Table 1.2). Finally, the *Menu* class represents all possible combinations of meals, enabling the creation of personalized menus tailored to the preferences or restrictions of a guest.

For: ● Meal	For: ● Ingredient	For: ● Menu
AranciniAlProsciutto	Apple	LactoseFreeGlutenFreeNoVegHighMenu
AranciniAlRagu	Avocado	LactoseFreeGlutenFreeNoVegLowMenu
AranciniVegetariani	Bacon	LactoseFreeGlutenFreeNoVegMediumMenu
ArrostatoCarne	Beef	LactoseFreeGlutenFreeVegHighMenu
ArrostatoPesce	Bread	LactoseFreeGlutenFreeVegLowMenu
Carbonara	Carrot	LactoseFreeGlutenFreeVegMediumMenu
GnocchiSugoPapera	Champignon	LactoseFreeGlutenNoVegHighMenu
GrigliataCarne	Clams	LactoseFreeGlutenNoVegLowMenu
GrigliataPesce	Coffee	LactoseFreeGlutenNoVegMediumMenu
Insalata	Duck	LactoseFreeGlutenVegHighMenu
InsalataRussa	Egg	LactoseFreeGlutenVegLowMenu
Macedonia	Eggplant	LactoseFreeGlutenVegMediumMenu

Figure 4.3: Lists with some of the objects of Ingredient and Meal.

Ingredient Class

The Ingredient class is used to represent the basic building blocks of meals. Each Ingredient object is characterized by four key **data properties** that define its attributes, as illustrated in Figure 4.4:

- *Has Lactose*: this property specifies whether an ingredient contains lactose or not. It has a range of *xsd:boolean*, allowing values of *true* or *false*. In the example shown in Figure 4.4, this property is set to *false*.
- *Is Vegetarian*: this property is used to indicate if an ingredient contains meat. Its range is *xsd:boolean*, with possible values of *true* or *false*. In the example in Figure 4.4, this property is also set to *false*.
- *Has Gluten*: this property indicates whether an ingredient contains gluten or is gluten-free. It also uses a range of *xsd:boolean*, meaning it can be either *true* or *false*. In the example in Figure 4.4, this property is set to *false*.
- *Has Name*: this property provides the name of the ingredient. Unlike the other three data properties, this one has a range of *xsd:string*, meaning it can hold any text value representing the ingredient's name. In the example in Figure 4.4, this property is set to "Beef".



Figure 4.4: Example of Ingredient object.

These data properties are utilized by different SWRL rules (discussed in Section 4.2) to categorize Ingredient objects into various sub classes, listed in Figure 4.5. For instance, the *beef* object (Figure 4.4), which has the property *isVegetarian* set to *false*, is classified into the *NoVegIngredient* subclass rather than the *VegIngredient* subclass. This is because they are both mutually exclusive subclasses of Ingredient. This assures that each ingredient is appropriately categorized.



Figure 4.5: Ingredient subclasses.

In addition to those data properties, the Ingredient class also has an **object property** called *isIn Meal*, as shown in Figure 4.4. This property has a domain of Ingredient and a range of Meal, indicating the relationship between a specific ingredient and the meals in which it appears. Since an ingredient can be part of multiple meals, this property can have multiple instances for each ingredient.

Meal Class

The Meal class represents individual dishes and their properties, as well as their relationships to ingredients. Each Meal object is characterized by five key **data properties**, which provide detailed information about each meal and are essential for categorizing and managing the meals within my system. These properties are illustrated in Figure 4.6:

- *Has Name*: This property specifies the name of the meal. The range is *xsd:string*, allowing it to hold any text value representing the meal's name. In the example shown in Figure 4.6, this property is set to "*Arancini vegetariani*".
- *Has Calories*: This property indicates the caloric level of the meal. The range is

xsd:string, which can represent various calorie levels, such as *Low*, *Medium*, or *High*. In the example shown in Figure 4.6, this property is set to *"Medium"*.

- *Mealls Vegetarian*: This property denotes whether the meal is vegetarian or not. The range is *xsd:boolean*, indicating whether the meal is suitable for vegetarians (true) or not (false). In the example shown in Figure 4.6, this property is set to *true*.
- *MealHasGluten*: This property specifies whether the meal contains gluten. The range is *xsd:boolean*, with true indicating the presence of gluten and false indicating its absence. In the example shown in Figure 4.6, this property is set to *true*.
- *MealHasLactose*: This property indicates whether the meal contains lactose. Its range is *xsd:boolean*, with true denoting the presence of lactose and false denoting its absence. In the example shown in Figure 4.6, this property is set to *true*.

Property assertions: AranciniVegetariani	
Object property assertions +	
hasIngredients	Onion
hasIngredients	Oil
hasIngredients	Salt
hasIngredients	Rice
hasIngredients	Water
hasIngredients	Flour
hasIngredients	Bread
hasIngredients	Mozzarella
hasIngredients	TomatoSauce
Data property assertions +	
MeallsVegetarian	true
MealHasGluten	true
hasName	"Arancini vegetariani"
hasCalories	"Medium"
MealHasLactose	true

Figure 4.6: Example of Meal object.

Similar to the categorization of ingredients using SWRL rules, meals are classified into distinct subclasses (see Figure 4.7) based on the properties *MeallsVegetarian*, *MealHasGluten*, *MealHasLactose* and *hasCalories*.

For example, consider the meal in Figure 4.6. Since this meal has the *MeallsVegetarian* property set to true, it is classified as suitable for vegetarians. Consequently, this meal would be categorized under the *VegMeal* subclass of *Meal* class. Conversely, it would be excluded from the *NoVegMeal* subclass, as these two subclasses are mutually disjoint.

This classification system allows to efficiently manage and filter meals based on dietary needs and preferences, ensuring that customers receive meal suggestions that align with their specific requirements.

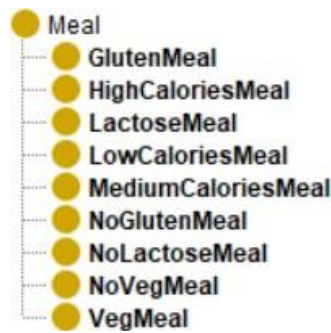


Figure 4.7: List of Meal subclasses.

In addition to these data properties, the Meal class includes an **object property** called *hasIngredients*, as shown in Figure 4.6. This property establishes a relationship between a meal and its ingredients, with a domain of Meal and a range of Ingredient. Since a meal can consist of multiple ingredients, this property can occur multiple times for a single meal. To complement this, the inverse property *isInMeal* is defined in the Ingredient class (see Section 4.1.1), creating a bidirectional link that connects ingredients back to the meals they belong to.

Menu Class

The Menu class is designed to represent personalized menus that cater to all kinds of client preferences. Unlike the Ingredient and Meal classes, the Menu class does not have any data properties. Instead, it is defined solely by an **object property** called *hasMeal*. This property establishes a relationship between a menu and the meals it contains, with a domain of Menu and a range of Meal.

Initially, the instances of the Menu class are empty placeholders. However, once the SWRL rules are executed, each Menu object is populated with the meals that satisfy the specific preferences provided by the client (see Figure 4.8). These preferences might include criteria such as lactose-free, gluten-free, vegetarian, or calorie-specific options.

Since a menu can contain multiple meals, the *hasMeal* property can have multiple instances for a single menu object. This allows the Menu class to accommodate a wide variety of combinations. For example, the Figure 4.8 demonstrates a menu object that, after applying SWRL rules, is populated with meals satisfying the preferences: lactose-free, gluten-free, non-vegetarian, and low calorie level.

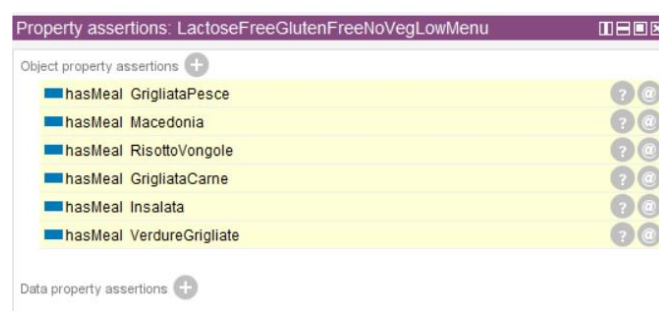


Figure 4.8: Example of a Menu object.

Unfortunately, the menus are currently restricted to meals that correspond exactly to the specified calorie preference. For instance, a menu with a high-calorie preference does not include meals categorized as medium or low-calorie, and a medium-calorie menu excludes low-calorie meals. Due to time constraints, I was unable to resolve this limitation during development.

4.2 SWRL Rules

SWRL (Semantic Web Rule Language) is a rule-based language that extends OWL (Web Ontology Language) by adding logic rules to ontologies. These rules have an antecedent (body) and a consequent (head) in the form of an implication. The rules state that whenever the conditions in the antecedent are true, the conditions in the consequent must also be true. Both parts can have multiple conditions, which are treated as conjunctions. SWRL rules allow for more advanced reasoning in knowledge systems, helping to make logical inferences based on the defined data [Swr].

I employed SWRL rules to categorize both ingredients and meals in my ontology. I introduced additional rules to dynamically populate the instances of the Menu class. Unlike the earlier rules, which permanently classify objects into subclasses thanks to the 'Drools-2OWL' button, these new rules dynamically assign meals to menus based on client preferences. However, their results are only visible during the execution of the reasoner Hermit 1.4.3.456.

Since many of the rules follow similar structures, we will focus on a few representative examples while listing the others for reference.

Vegetarian Ingredient Rule

This rule identifies vegetarian ingredients and classifies them into the *VegIngredient* subclass if there is *Vegetarian* property is set to true (the results of this rule are shown in Figure 4.9). Similarly, ingredients are classified into the *Gluten Ingredient* and *Lactose Ingredient* subclasses based on whether their *has Gluten* and *has Lactose* properties are set to true.

$$cintioni : Ingredient(?x) \wedge cintioni : isVegetarian(?x, true) \rightarrow cintioni : vegIngredient(?x).$$


Figure 4.9: Result of the vegetarian ingredient rule.

Non-vegetarian Ingredient Rule

This rule identifies non-vegetarian ingredients and classifies them into the *NoVegIngredient* subclass if there is *Vegetarian* property is set to false (the results of this rule are shown in Figure 4.10). A similar approach is used to classify gluten-free and lactose-free ingredients into their respective subclasses.

$$cintioni : Ingredient(?x) \wedge cintioni : isVegetarian(?x, false) \rightarrow cintioni : NoVegIngredient(?x).$$


Figure 4.10: Result of the non-vegetarian ingredient rule.

Vegetarian meals Rule

This rule classifies meals as vegetarian and places them in the *VegMeal* subclass if their *Meals Vegetarian* property is set to true (the results of this rule are shown in Figure 4.11). Similarly, meals can be classified as *NoGluten Meal* and/or *NoLactose Meal* if their respective properties are set to false.

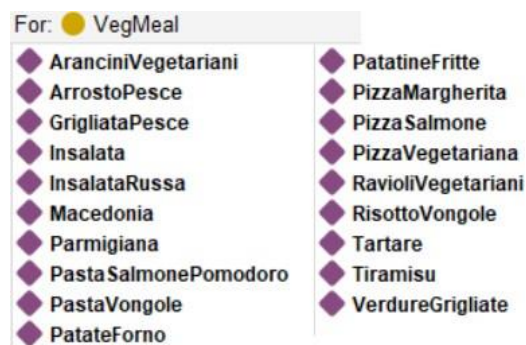
$$cintioni : Meal(?x) \wedge cintioni : MealsVegetarian(?x, true) \rightarrow cintioni : VegMeal(?x).$$


Figure 4.11: Result of the vegetarian meals rule.

Non-vegetarian meals Rule

This rule classifies a meal as *NoVeg Meal* if it contains any ingredient classified as *NoVeg Ingredient*. Specifically, if an ingredient *?x* is classified as *NoVeg Ingredient* and is part of a meal *?y*, as indicated by the property *isIn Meal*, then the meal *?y* is categorized as *NoVeg Meal* (the results of this rule are shown in Figure 4.12). Similarly, meals are classified into subclasses such as *GlutenMeal* or *LactoseMeal* if they contain any ingredient classified as *Gluten Ingredient* or *Lactose Ingredient*, respectively.

cintioni : *No vegIngredient*(?x) \wedge *cintioni* : *isIn Meal*(?x, ?y) \rightarrow *cintioni* : *NoV eg Meal*(?y).



Figure 4.12: Result of the non-vegetarian meals rule.

Low calories meals rule

This rule classifies a meal as *Low Calories Meal* if its *has Calories* property is set to "Low" (the results of this rule are shown in Figure 4.13). Similarly, there are rules to categorize meals into *Medium Calories Meal* and *High Calories Meal* based on their respective calorie levels.

cintioni : *Meal*(?x) \wedge *cintioni* : *hasCalories*(?x, "Low") \rightarrow *cintioni* : *LowCaloriesMeal*(?x).



Figure 4.13: Result of the low calories meals rule.

Reverse of hasIngredients rule

This rule defines the reverse relationship of the *hasIngredients* property. If a meal ?x has an ingredient ?y (indicated by the *hasIngredients* property), then the ingredient ?y is part of that meal ?x (indicated by the *isIn Meal* property). Figure 4.14 shows the results of this rule for the *Flour* object of the class *Ingredient*.

cintioni : *hasIngredients*(?x, ?y) \rightarrow *cintioni* : *isIn Meal*(?y, ?x).

Property assertions: Flour	
Object property assertions +	
isInMeal Pizza Salmone	isInMeal RavioliRagu
isInMeal GnocchiSugoPapera	isInMeal AranciniAlRagu
isInMeal PizzaVegetariana	isInMeal PizzaNorcina
isInMeal PizzaPatateBacon	isInMeal ScaloppineFunghi
isInMeal PizzaZucchineSalsiccia	isInMeal PizzaBoscaiola
isInMeal AranciniVegetariani	isInMeal PizzaMelanzaneProsciutto
isInMeal AranciniAlProsciutto	isInMeal PizzaMargherita
isInMeal PizzaProsciutto	isInMeal RavioliVegetariani

Figure 4.14: Result of the reverse of hasIn gredients rule for the Ingredient object *Flour*.

Lactose gluten non vegetarian high calories level menu rule

This rule populates the *Lactose GlutenNoVegHighMenu* instance of the *Menu* class with meals that satisfy specific conditions. It checks for meals (*?meal*) with the data property *has Calories* set to "High". Since this is the only restriction, if this conditions is met, the *has Meal* object property links the identified meals to the *LactoseGlutenNoVegHighMenu*.

Similar rules were defined for menus corresponding to the same preferences but with medium or low calorie levels. These rules follow the same structure, with the only difference being the value specified in the *has Calories* property: "Medium" for the *LactoseGlutenNoVegMediumMenu* and "Low" for the *LactoseGlutenNoVegLowMenu*.

Figure 4.15 provides an example of this rule in action, illustrating how meals with an high calorie levels are assigned to this specific menu.

$$\begin{aligned} & \text{cintioni} : \text{Menu}(\text{cintioni} : \text{LactoseGlutenNoV egHighMenu}) \wedge \text{cintioni} : \\ & \text{Meal}(\text{?meal}) \wedge \text{cintioni} : \text{hasCalories}(\text{?meal}, \text{"High"}) \rightarrow \text{cintioni} : \\ & \text{hasMeal}(\text{cintioni} : \text{LactoseGlutenNoV egHighMenu}, \text{?meal}). \end{aligned}$$

Property assertions: LactoseGlutenNoVegHighMenu	
Object property assertions +	
hasMeal PizzaPatateBacon	
hasMeal PizzaBoscaiola	
hasMeal Parmigiana	
hasMeal Carbonara	
hasMeal RavioliRagu	
hasMeal InsalataRussa	
hasMeal GnocchiSugoPapera	
hasMeal AranciniAlRagu	
hasMeal PizzaProsciutto	
hasMeal Tartare	
hasMeal PizzaNorcina	
hasMeal Tiramisu	
hasMeal Tagliere	
hasMeal PatatineFritte	
hasMeal PizzaMelanzaneProsciutto	
hasMeal PizzaZucchineSalsiccia	
hasMeal MelanzaneRipiene	

Figure 4.15: Result of the lactose gluten non vegetarian high calories level menu rule.

Lactose-free gluten-free vegetarian low calories level menu rule

This rule populates the *LactoseFreeGlutenFreeVegLowMenu* instance of the Menu class with meals that meet a comprehensive set of dietary preferences and calorie restrictions. Specifically, it selects meals (*?meal*) that satisfy the following conditions:

- They do not contain lactose: *MealHasLactose* = *false*.
- They do not contain gluten: *MealHasGluten* = *false*.
- They are vegetarian: *MeallsVegetarian* = *true*.
- They have a low calorie level: *hasCalories* = "Low".

Once a meal meets all these conditions, it is associated with the *LactoseFreeGlutenFreeVegLowMenu* instance through the *has Meal* property.

```
cintioni : Menu(cintioni : LactoseFreeGlutenFreeVegLowMenu) ∧ cintioni :
    Meal(?meal) ∧ cintioni : MealHasLactose(?meal, false) ∧ cintioni :
        MealHasGluten(?meal, false) ∧ cintioni :
            MeallsVegetarian(?meal, true) ∧ cintioni : hasCalories(?meal, "Low") – >
cintioni : hasMeal(cintioni : LactoseFreeGlutenFreeVegLowMenu, ?meal).
```

This rule represents the most restrictive case, as it considers all potential constraints simultaneously (the results are shown in Figure 4.16). However, the other rules used to populate the different kinds of menus are designed to adapt dynamically to varying preferences by omitting conditions that are not required. Specifically:

- If the meals can have lactose, the *MealHasLactose(?meal, false)* condition is excluded.
- If the meal do not have to be vegetarian, the *MeallsVegetarian(?meal, true)* condition is removed.
- If the meal can contain gluten, the *MealHasGluten(?meal, false)* condition is removed.
- Like we already said in the previous rule, if the calorie level is different it will be sufficient to change the "Low" value in the *hasCalories(?meal, "Low")* condition.

Property assertions: LactoseFreeGlutenFreeVegLowMenu	
Object property assertions +	
hasMeal	GrigliataPesce
hasMeal	Macedonia
hasMeal	RisottoVongole
hasMeal	Insalata
hasMeal	VerdureGrigliate

Figure 4.16: Result of the lactose-free gluten-free vegetarian low calories level menu rule.

4.3 SHACL Shapes

SHACL (Shapes Constraint Language) is used to validate RDF graphs by defining conditions through shapes expressed as RDF graphs. These "shapes graphs" specify constraints that data graphs must meet, ensuring data validity. SHACL shapes facilitate data validation, user interface development, and integration by providing structured criteria for RDF data [Sha].

In our ontology we defined different shacl shapes:

- *Ingredient Shape*: Ensures that each Ingredient has a single name and defines its attributes regarding gluten, lactose, and vegetarian status. Each of these properties must be present and of the correct type. Additionally, if an Ingredient is associated with the *isInMeal* property, it must be linked to a meal.

```

1 ex:IngredientShape
2   a sh:NodeShape ;
3   sh:targetClass ex:Ingredient ; # Applies to all ingredients
4   sh:property [
5     sh:path ex:hasName ; #add constraints to hasName property
6     sh:datatype xsd:string ;
7     sh:minCount 1 ;
8     sh:maxCount 1 ;
9     sh:severity sh:Warning ;
10  ] ;
11  sh:property [
12    sh:path ex:hasGluten ; #add constraints to hasGluten property
13    sh:datatype xsd:boolean ;
14    sh:minCount 1 ;
15    sh:maxCount 1 ;
16    sh:severity sh:Warning ;
17  ] ;
18  sh:property [
19    sh:path ex:hasLactose ; #add constraints to hasLactose
20    sh:datatype xsd:boolean ;
21    sh:minCount 1 ;
22    sh:maxCount 1 ;
23    sh:severity sh:Warning ;
24  ] ;
25  sh:property [
26    sh:path ex:isVegetarian ; #add constraints to isVeegetarian
27    sh:datatype xsd:boolean ;
28    sh:minCount 1 ;
29    sh:maxCount 1 ;
30    sh:severity sh:Warning ;
31  ] ;
32  sh:property [
33    sh:path ex:isInMeal ; #add constraints to isInMeal property
34    sh:class ex:Meal ;
35    sh:severity sh:Warning ;
36  ] .
37

```

Listing 4.1: Ingredient Shape

If any of these properties does not satisfy its constraints a warning will be returned.

- **GlutenIngredientShape**: Enforces that GlutenIngredient instances must have the *hasGluten* property set to true.

```

1 ex:GlutenIngredientShape
2   a sh:NodeShape ;
3   sh:targetClass ex:GlutenIngredient ; # Applies to all gluten
  ingredients
4   sh:property [
5     sh:path ex:hasGluten ; #add constraints to hasGluten property
6     sh:hasValue true;
7   ].
8

```

Listing 4.2: GlutenIngredientShape

If the *hasGluten* property has value *false*, an error will occur.

- **NoGlutenIngredientShape**: Enforces that NoGlutenIngredient instances must have the *hasGluten* property set to false.

```

1   ex:NoGlutenIngredientShape
2   a sh:NodeShape ;
3   sh:targetClass ex:NoGlutenIngredient ; # Applies to all non-gluten
  ingredients
4   sh:property [
5     sh:path ex:hasGluten ; #add constraints to hasGluten property
6     sh:hasValue false;
7   ].
8

```

Listing 4.3: NoGlutenIngredientShape

Similarly, if the *hasGluten* property has value *true*, an error will occur.

- **LactoseIngredientShape**: Enforces that LactoseIngredient instances must have the *hasLactose* property set to true.

```

1 ex:LactoseIngredientShape
2   a sh:NodeShape ;
3   sh:targetClass ex:LactoseIngredient ; # Applies to all lactose
  ingredients
4   sh:property [
5     sh:path ex:hasLactose ; #add constraints to hasLactose
  property
6     sh:hasValue true;
7   ].
8

```

Listing 4.4: LactoseIngredient Shape

Similarly, if the *hasLactose* property has value *false*, an error will occur.

- **NoLactoseIngredientShape**: Enforces that NoLactoseIngredient instances must have the *hasLactose* property set to false.

```

1 ex:NoLactoseIngredientShape
2   a sh:NodeShape ;
3   sh:targetClass ex:NoLactoseIngredient ; # Applies to all non-
  lactose ingredients
4   sh:property [
5     sh:path ex:hasLactose ; #add constraints to hasLactose
  property

```

```

6       sh:hasValue false;
7   ].
8

```

Listing 4.5: NoLactoseIngredientShape

Similarly, if the *has Lactose* property has value *true*, an error will occur.

- **VegIngredientShape:** Ensures VegIngredient instances have the *is Vegetarian* property set to true.

```

1 ex:VegIngredientShape
2   a sh:NodeShape ;
3   sh:targetClass ex:VegIngredient ; # Applies to all Vegetarian
  ingredients
4   sh:property [
5     sh:path ex:isVegetarian ; #add constraints to isVegetarian
    property
6     sh:hasValue true;
7   ].
8

```

Listing 4.6: VegIngredient Shape

Similarly, if the *is Vegetarian* property has value *false*, an error will occur.

- **NoVegIngredientShape:** Ensures NoVegIngredient instances have the *is Vegetarian* property set to false.

```

1 ex:NoVegIngredientShape
2   a sh:NodeShape ;
3   sh:targetClass ex:NoVegIngredient ; # Applies to all non-vegetarian
  ingredients
4   sh:property [
5     sh:path ex:isVegetarian ; #add constraints to isVegetarian
    property
6     sh:hasValue false;
7   ].
8

```

Listing 4.7: NoVegIngredientShape

Similarly, if the *is Vegetarian* property has value *true*, an error will occur.

- **Meal Shape:** Ensures that each Meal has a single name and defines its attributes related to gluten, lactose, vegetarian status, and calorie level. Each of these properties must be present and of the correct type. Additionally, if a meal includes the *has Ingredients* property, it must be associated with a valid ingredient.

```

1 ex:MealShape
2   a sh:NodeShape ;
3   sh:targetClass ex:Meal ; # Applies to all meals
4   sh:property [
5     sh:path ex:hasName ; # Add constraints to hasName property
6     sh:datatype xsd:string ;
7     sh:minCount 1 ;
8     sh:maxCount 1 ;
9     sh:severity sh:Warning ;
10  ] ;
11  sh:property [
12    sh:path ex:MealHasGluten ; # Add constraints to MealHasGluten
    property

```

```

13     sh:datatype xsd:boolean ;
14     sh:minCount 1 ;
15     sh:maxCount 1 ;
16     sh:severity sh:Warning ;
17   ] ;
18   sh:property [
19     sh:path ex:MealHasLactose ; # Add constraints to
MealHasLactose property
20     sh:datatype xsd:boolean ;
21     sh:minCount 1 ;
22     sh:maxCount 1 ;
23     sh:severity sh:Warning ;
24   ] ;
25   sh:property [
26     sh:path ex:MealIsVegetarian ; # Add constraints to
MealIsVegetarian property
27     sh:datatype xsd:boolean ;
28     sh:minCount 1 ;
29     sh:maxCount 1 ;
30     sh:severity sh:Warning ;
31   ] ;
32   sh:property [
33     sh:path ex:hasCalories ; # Add constraints to hasCalories
property
34     sh:datatype xsd:string ;
35     sh:minCount 1 ;
36     sh:maxCount 1 ;
37     sh:severity sh:Warning ;
38   ] ;
39   sh:property [
40     sh:path ex:hasIngredients ; # Add constraints to
hasIngredients property
41     sh:class ex:Ingredient ;
42     sh:severity sh:Warning ;
43   ].
44

```

Listing 4.8: Meal Shape

If any of these properties does not satisfy its constraints a warning will be returned.

- *GlutenMealShape*: Enforces that instances of *GlutenMeal* have the *MealHasGluten* property set to true.

```

1 ex:GlutenMealShape
2   a sh:NodeShape ;
3   sh:targetClass ex:GlutenMeal ; # Applies to all gluten meals
4   sh:property [
5     sh:path ex:MealHasGluten ; # Add constraints to MealHasGluten
property
6     sh:hasValue true ;
7   ].
8

```

Listing 4.9: Gluten Meal Shape

If the *Meal has Gluten* property has value *false*, an error will occur.

- *NoGlutenMealShape*: Ensures that instances of *NoGlutenMeal* have the *MealHasGluten* property set to false and that all its ingredients are gluten-free.

```

1 ex:NoGlutenMealShape
2   a sh:NodeShape ;
3   sh:targetClass ex:NoGlutenMeal ; # Applies to all non-gluten meals
4   sh:property [
5     sh:path ex:MealHasGluten ; # Add constraints to MealHasGluten
    property
6     sh:hasValue false ;
7   ];
8   sh:property [
9     sh:path ex:hasIngredients ; # Ensures all ingredients in a
    NoGlutenMeal are gluten-free
10    sh:node [
11      sh:path ex:hasGluten ;
12      sh:hasValue false ;
13    ] ;
14  ].
15

```

Listing 4.10: NoGlutenMealShape

If the *Meal Has Gluten* property has value *true*, or if any of the ingredients in the meal contain gluten, an error will occur.

- **Lactose Meal Shape:** Enforces that instances of Lactose Meal have the *Meal Has Lactose* property set to true.

```

1 ex:LactoseMealShape
2   a sh:NodeShape ;
3   sh:targetClass ex:LactoseMeal ; # Applies to all lactose meals
4   sh:property [
5     sh:path ex:MealHasLactose ; # Add constraints to
    MealHasLactose property
6     sh:hasValue true ;
7   ].
8

```

Listing 4.11: LactoseMealShape

If the *Meal Has Lactose* property has value *false*, an error will occur.

- **NoLactoseMealShape:** Ensures that instances of NoLactoseMeal have the *Meal-HasLactose* property set to false and that all its ingredients are lactose-free.

```

1 ex:NoLactoseMealShape
2   a sh:NodeShape ;
3   sh:targetClass ex:NoLactoseMeal ; # Applies to all non-lactose
    meals
4   sh:property [
5     sh:path ex:MealHasLactose ; # Add constraints to
    MealHasLactose property
6     sh:hasValue false ;
7   ];
8   sh:property [
9     sh:path ex:hasIngredients ; # Ensures all ingredients in a
    NoLactoseMeal are lactose-free
10    sh:node [
11      sh:path ex:hasLactose ;
12      sh:hasValue false ;
13    ] ;
14  ].
15

```

Listing 4.12: NoLactose Meal Shape

If the *Meal Has Lactose* property has value *true*, or if any of the ingredients in the meal contain lactose, an error will occur.

- **Veg Meal Shape:** Ensures that instances of Veg Meal have the *Mealls Vegetarian* property set to true and that all ingredients are vegetarian.

```

1 ex:VegMealShape
2   a sh:NodeShape ;
3   sh:targetClass ex:VegMeal ; # Applies to all vegetarian meals
4   sh:property [
5     sh:path ex:MealIsVegetarian ; # Add constraints to
MealIsVegetarian property
6     sh:hasValue true ;
7   ];
8   sh:property [
9     sh:path ex:hasIngredients ; # Ensures all ingredients in a
NoGlutenMeal are gluten-free
10    sh:node [
11      sh:path ex:isVegetarian ;
12      sh:hasValue true ;
13    ] ;
14  ].
15

```

Listing 4.13: Veg Meal Shape

If the *Mealls Vegetarian* property has value *false*, or if any of the ingredients in the meal are not vegetarian, an error will occur.

- **NoVegMealShape:** Ensures that instances of NoVegMeal have the *MeallsVegetarian* property set to false.

```

1 ex:NoVegMealShape
2   a sh:NodeShape ;
3   sh:targetClass ex:NoVegMeal ; # Applies to all non-vegetarian meals
4   sh:property [
5     sh:path ex:MealIsVegetarian ; # Add constraints to
MealIsVegetarian property
6     sh:hasValue false ;
7   ].
8

```

Listing 4.14: NoVegMealShape

If the *Mealls Vegetarian* property has value *true*, an error will occur.

- **Low Calories Meal Shape:** Enforces that instances of Low Calories Meal have the *Has Calories* property set to "Low".

```

1 ex:LowCaloriesMealShape
2   a sh:NodeShape ;
3   sh:targetClass ex:LowCaloriesMeal ; # Applies to all low calories
meals
4   sh:property [
5     sh:path ex:hasCalories ; # Add constraints to hasCalories
property
6     sh:hasValue "Low" ;
7   ].
8

```

Listing 4.15: LowCaloriesMealShape

If the *has Calories* property has not value "Low", an error will occur.

- **Medium Calories Meal Shape:** Ensures that instances of Medium Calories Meal have the *has Calories* property set to "Medium".

```

1 ex:MediumCaloriesMealShape
2   a sh:NodeShape ;
3   sh:targetClass ex:MediumCaloriesMeal ; # Applies to all medium
   calories meals
4   sh:property [
5     sh:path ex:hasCalories ; # Add constraints to hasCalories
   property
6     sh:hasValue "Medium" ;
7   ].
8

```

Listing 4.16: Medium Calories Meal Shape

If the *has Calories* property has not value "Medium", an error will occur.

- **High Calories Meal Shape:** Ensures that instances of High Calories Meal have the *Has Calories* property set to "High".

```

1 ex:HighCaloriesMealShape
2   a sh:NodeShape ;
3   sh:targetClass ex:HighCaloriesMeal ; # Applies to all high calories
   meals
4   sh:property [
5     sh:path ex:hasCalories ; # Add constraints to hasCalories
   property
6     sh:hasValue "High" ;
7   ].
8

```

Listing 4.17: High Calories Meal Shape

If the *has Calories* property has not value "High", an error will occur.

- **Menu Shape:** ensures that instances of the Menu class have the *has Meal* property pointing to valid instances of the Meal class.

```

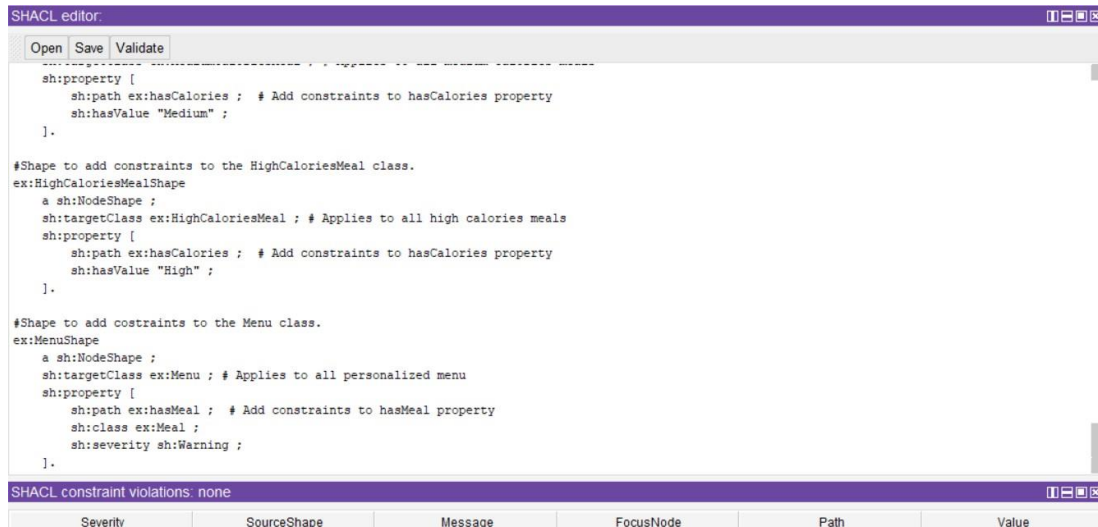
1 ex:MenuShape
2   a sh:NodeShape ;
3   sh:targetClass ex:Menu ; # Applies to all personalized menu
4   sh:property [
5     sh:path ex:hasMeal ; # Add constraints to hasMeal property
6     sh:class ex:Meal ;
7     sh:severity sh:Warning ;
8   ].
9

```

Listing 4.18: Menu Shape

If the *has Meal* property does not have a valid Meal object a warning will be returned.

The result given by validating the SHACL Shapes file (*shacl shapes*) is shown in Figure 4.17 and proves that no constraint is violated.

Figure 4.17: Result of the validation of the *shacl shapes* file.

4.4 SPARQL Queries

SPARQL is a specification for querying and manipulating RDF data, offering a language and protocol for retrieving and updating information in RDF graphs on the Web or within RDF stores [Spa]. In my project, I used Graph DB [Ont], a powerful RDF database that supports SPARQL queries and provides advanced features for managing and querying large-scale RDF datasets.

I conduct numerous queries; however, many are similar, differing primarily by the class they target. Therefore, I will present only one representative example. Additionally, I will present the query that displays the project's final result, showing the meals chosen by the client. Figure 4.18 shows the prefixes we used to simplify and clarify the queries. Prefixes are shorthand representations for URIs that make queries more readable by allowing the use of short names.

```

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ex: <http://www.semanticweb.org/kebi-exam/2023-2024/cintioni.chiara-falcone.denise#>

```

Figure 4.18: Queries prefixes.

Query to get all elements of a class

The query in Figure 4.19 serves as an example of how to retrieve all elements of a specific class from the ontology. In this case, it fetches all instances of the *ex:NoVegIn gredient* class (see Figure 4.20), which represents non-vegetarian ingredients. By querying for the *rdf:type* of *ex:NoVegIn gredient*, it returns all ingredients classified as non-vegetarian. This approach can be generalized for any class defined in the ontology. By replacing *ex:NoVegIn gredient* with the target class name, the query can be adapted to retrieve all instances of the specified class.

```
#Query to get all non-vegetarian ingredients
SELECT ?ingredient
WHERE {
  ?ingredient rdf:type ex:NoVegIngredient .
}
```

Figure 4.19: Query to get all the non-vegetarian ingredients.

1	ex:Ham
2	ex:Beef
3	ex:Sausage
4	ex:Bacon
5	ex:Duck

Figure 4.20: Result of the query in Figure 4.19.

Query to get only the preferred meals

The query shown in Figure 4.21 retrieves meals that match a client's preferences by applying filters based on lactose, gluten, vegetarian status, and calorie level:

- *Lactose and Gluten*: The query uses *BIND* to set flags for lactose and gluten intolerances. If the client is not intolerant (*?check Lactose = false* and *?check Gluten = false*), it includes all meals regardless of these attributes. If the client is intolerant, only meals that do not contain lactose or gluten are selected.
- *Vegetarian Preference*: Similarly, the query sets a flag to check if the client prefers vegetarian meals. If *?check Vegetarian = true*, only vegetarian meals are included.
- *Calorie Level*: The query filters meals based on the calorie level according to the client's preference. If the client specifies "Low", only low-calorie meals are returned. If the client prefers "Medium", the query returns both low and medium-calorie meals. If the client selects "High" or leaves empty "", all meals are included.

In Figure 4.22 we can see the result of the query in Figure 4.21. In this case, the client does not have any allergies and is not vegetarian. Therefore, the only constraint applied is based on the calorie level, with *?calories Level* set to "Low". This means that the query will return only the meals that have a calorie level classified as "Low".


```

SELECT DISTINCT ?mealName WHERE {
  BIND(false AS ?checkLactose) # Set this to true if allergic to lactose
  BIND(false AS ?checkGluten) # Set this to true if allergic to gluten
  BIND(false AS ?checkVegetarian) # Set this to true if vegetarian
  BIND("Low" AS ?caloriesLevel) # Set this to "Low", "Medium", "High" or "" if all is fine.

  ?meal rdf:type ex:Meal ;
    ex:hasName ?mealName ;
    ex:MealIsVegetarian ?isVegetarian;
    ex:MealHasGluten ?hasGluten;
    ex:MealHasLactose ?hasLactose.

  # Lactose filter
  FILTER (?checkLactose = false || NOT EXISTS {
    ?meal rdf:type ex:LactoseMeal .
  })

  # Gluten filter
  FILTER (?checkGluten = false || NOT EXISTS {
    ?meal rdf:type ex:GlutenMeal.
  })

  # Vegetarian filter
  FILTER (?checkVegetarian = false || NOT EXISTS {
    ?meal rdf:type ex:NoVegMeal.
  })

  # Calories filter
  FILTER (
    ?caloriesLevel = "" || ?caloriesLevel = "High" ||
    (
      (?caloriesLevel = "Low" && EXISTS { ?meal rdf:type ex:LowCaloriesMeal }) ||
      (?caloriesLevel = "Medium" && NOT EXISTS { ?meal rdf:type ex:HighCaloriesMeal })
    )
  )
}

```

Figure 4.21: Query that returns the meals that match the client's preferences.

1	"Grigliata di carne"
2	"Grigliata di pesce"
3	"Insalata"
4	"Macedonia"
5	"Pasta salmone e pomodoro"
6	"Pasta alle vongole"
7	"Risotto alle vongole"
8	"Scaloppine ai funghi"
9	"Verdure grigliate"

Figure 4.22: Result of the final query if the client doesn't have any allergies, is not vegetarian but only wants low calorie meals.

5. Agile and Ontology-based Meta-modelling

This chapter delves into the innovative approach of agile and ontology-based meta-modeling, a methodology designed to enhance the adaptability and relevance of modeling languages across various domains. By blending agile development principles with ontology-based frameworks, this approach offers a flexible and dynamic solution for designing and refining complex models.

5.1 AOAME

To work on the second task of the project, which involved agile and ontology-based meta-modeling, we utilized AOAME [and19]. This tool is designed to support the creation and management of Enterprise Knowledge Graph (EKG) schemas, enabling the organization of domain-specific knowledge for effective analysis, reasoning, and integration across various data sources. AOAME facilitates dynamic modification and management of modeling constructs through meta-modeling operators that generate SPARQL queries and update the triple store, ensuring consistency between the model and the data. Built with Jena Fuseki Java libraries, AOAME is well-suited for adapting to real-world needs and supporting flexible meta-modeling. It played a key role in customizing BPMN 2.0 by allowing us to extend its existing classes and properties to meet our specific requirements.

BPMN

BPMN (Business Process Model and Notation) 2.0 is a standard for modeling business processes in a graphical format. It provides a comprehensive notation that is easy to understand and implement, facilitating communication about these procedures both within and between organizations, enhancing the ability to manage performances, collaborations, and transactions. It includes elements such as tasks, events, gateways, and flows to represent process workflows, decision points, and interactions. Its goal is to bridge the gap between business process design and execution by providing a standardized method for documenting and analyzing business processes [Bpm].

5.1.1 BPMN model

The Figure 5.1 presents the BPMN 2.0 model created with AOAME. This model visualizes the process of generating a personalized menu for a client. The model is organized into two separate pools: one for the Client and one for the System, clearly distinguishing their interactions.

The process begins when the client scans a QR code to initiate the workflow. The client then enters their preferences—such as dietary restrictions and allergies—and submits them to the system. Once the system receives the preferences, it passes the information to an extended class, *Create Personalized Menu*, which is specifically designed to select meals that match the client’s preferences. After the menu is personalized, the system sends it back to the client. The process concludes when the client receives their customized menu.

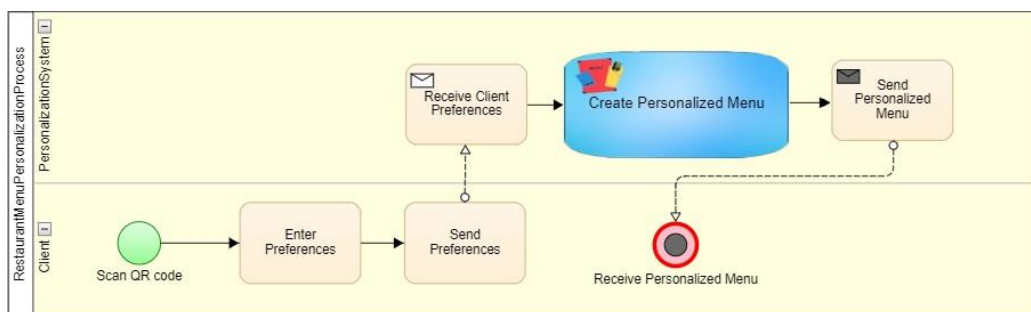


Figure 5.1: BPMN model that describes the process of personalization of a menu in a restaurant.

The extended class *Create Personalized Menu* is an extended class of *Task* and includes the data properties shown in Figure 5.2. The properties *isVegetarian*, *isGlutenIntolerant*, and *isLactoseIntolerant* are all of boolean type, while *preferredCaloriesLevel* is a string that can take the values "Low", "Medium", or "High".

Model element attributes

ID: CreatePersonalizedMenu_b7fa7303-5684-48d4-9c8d-03572c576922
Instantiation Type: Instance

Relation	Value	Actions
isVegetarian	<input type="text" value="false"/>	<button>Remove</button>
isGlutenIntolerant	<input type="text" value="true"/>	<button>Remove</button>
isLactoseIntolerant	<input type="text" value="true"/>	<button>Remove</button>
preferredCaloriesLevel	<input type="text" value="High"/>	<button>Remove</button>

lo:preferredCaloriesLevel Add Relation

Figure 5.2: Example of the properties for Create Personalized Menu where the client is intolerant to lactose and gluten.

5.2 Jena Fuseki

Jena Fuseki is a SPARQL server and a component of the Apache Jena framework, designed for hosting RDF (Resource Description Framework) data and providing a platform for querying and managing this data using SPARQL. Fuseki supports both SPARQL query (for reading data) and SPARQL update (for modifying data) operations. It can run as a standalone server, offering HTTP endpoints for querying and updating RDF data, or be embedded in applications, making it a versatile tool for developing semantic web applications and knowledge graph systems [Jen].

5.2.1 Query in Jena Fuseki

On the Jena Fuseki server, I defined a query (see Figure 5.3) that retrieves the personalized menu from the ontology created in Chapter 4 based on the model defined in AOAME. This query leverages the extended class and its properties that were specifically designed for this purpose, allowing for the extraction of meals tailored to the client's preferences. This query functions similarly to the one described in Section 4.4, with the main differences being the use of additional prefixes and the retrieval of preferences directly from the extended class rather than defining them within the query itself. In Figure 5.4 we can see the results of the preferences passed in Figure 5.2.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ex: <http://www.semanticweb.org/kebi-exam/2023-2024/cintioni.chiara-falcone.denise#>
PREFIX bpaas: <http://ikm-group.ch/archimeo/bpaas#>
PREFIX mod: <http://fhmw.ch/modelingEnvironment/ModelOntology#>
PREFIX lo: <http://fhmw.ch/modelingEnvironment/LanguageOntology#>

#Query to get only the preferred meals
SELECT DISTINCT ?mealName WHERE {
  mod:CreatePersonalizedMenu_b7fa7303-5684-48d4-9c8d-03572c576922 lo:isVegetarian ?checkVegetarian. #check isVegetarian value
  mod:CreatePersonalizedMenu_b7fa7303-5684-48d4-9c8d-03572c576922 lo:isLactoseIntolerant ?checkLactose. # check isLactoseIntolerant value
  mod:CreatePersonalizedMenu_b7fa7303-5684-48d4-9c8d-03572c576922 lo:isGlutenIntolerant ?checkGluten. # check isGlutenIntolerant value
  mod:CreatePersonalizedMenu_b7fa7303-5684-48d4-9c8d-03572c576922 lo:preferredCaloriesLevel ?caloriesLevel. # check preferredCaloriesLevel value

  ?meal rdf:type ex:Meal ;
    ex:hasName ?mealName ;
    ex:MealIsVegetarian ?isVegetarian;
    ex:MealHasGluten ?hasGluten;
    ex:MealHasLactose ?hasLactose.

  # Lactose filter
  FILTER (?checkLactose = false || NOT EXISTS {
    ?meal rdf:type ex:LactoseMeal .
  })

  # Gluten filter
  FILTER (?checkGluten = false || NOT EXISTS {
    ?meal rdf:type ex:GlutenMeal.
  })

  # Vegetarian filter
  FILTER (?checkVegetarian = false || NOT EXISTS {
    ?meal rdf:type ex:NoVegMeal.
  })

  # Calories filter
  FILTER (
    ?caloriesLevel = "" || ?caloriesLevel = "High" ||
    (
      (?caloriesLevel = "Low" && EXISTS { ?meal rdf:type ex:LowCaloriesMeal }) ||
      (?caloriesLevel = "Medium" && NOT EXISTS { ?meal rdf:type ex:HighCaloriesMeal })
    )
  )
}
```

Figure 5.3: Query to get the preferred meals only.

mealName	
1	Macedonia
2	Arrosto di carne
3	Arrosto di pesce
4	Tartare
5	Grigliata di carne
6	Insalata
7	Insalata russa
8	Risotto salsiccia e funghi
9	Risotto alle vongole
10	Verdure grigliate
11	Grigliata di pesce
12	Patatine fritte
13	Patate al forno

Showing 1 to 13 of 13 entries

Figure 5.4: Result of the query in Figure 5.3 with the values in Figure 5.2.

6. Conclusion

In this project, I developed a personalized menu system based on user preferences using various technologies and knowledge representation techniques. The goal was to represent a client dietary preferences and restrictions, such as lactose-free, gluten-free, vegetarian, and calorie levels, and use this information to dynamically generate personalized menus. The project involved working with several tools, including decision tables in Camunda DMN, Prolog for logical reasoning, Protégé for ontology creation, SHACL Shapes and SWRL Rules for enforcing constraints and managing the ontology. Additionally, we integrated SPARQL for querying and used GraphDB for data storage. Furthermore, Jena Fuseki and AOAME played key roles in managing RDF data and supporting agile ontology-based meta-modeling, allowing us to build and maintain complex, flexible models.

Below, I summarize the key aspects of the project, focusing on what we achieved and the challenges faced, followed by potential future work.

Decision Tables in Camunda DMN

Decision tables in Camunda DMN are composed of rows representing rules, each containing a set of conditions that must be met to produce a corresponding output. When multiple rules are satisfied for a given input, the Hit Policy determines the appropriate output. In my implementation, I employed the *Collect* policy, which aggregates and returns all outputs where rules are satisfied. This approach allowed me to generate comprehensive lists of ingredients and meals tailored to specific inputs, facilitating a decision-making process that is both clear and effective.

The inputs were structured to map user preferences to ingredient characteristics—such as lactose-free, gluten-free, and vegetarian—and to meal characteristics like calorie levels. Following my initial submission, I simplified my decision table model to ensure it adhered to its intended purpose as a decision-making tool, rather than functioning like a scripted process. To achieve this, I revised the logic of the model, restructuring the *Final Meals* table to include a rule for each possible input configuration along with the corresponding meal outputs.

Advantages and Highlights

I found decision tables to be an excellent tool for implementing decision-making logic due to their straightforward and clear structure. The row-based format, where each rule consists of defined conditions and corresponding outputs, made the system highly intuitive and user-friendly, especially when designing and refining the model. This clarity significantly helped when I had to implement and modify the tables.

The feature I appreciated the most was the *Collect* policy, which allowed me to return multiple results—such as lists of ingredients and meals—based on user preferences. This policy proved especially helpful, as my project required returning multiple items, and initially, I was unsure how to achieve this. However, as I became more familiar with the system and the nuances of the different policies, this policy made implementation smooth and efficient. This understanding enabled me to transition from my initial model to a more streamlined and effective one in a relatively short timeframe.

Challenges and Issues

The most significant challenge i encountered with decision tables and Camunda DMN was addressing the issues with my initial model following the first submission. Specifically, i had to understand why the previous model was behaving more like a scripted solution rather than a proper decision table. This required a deep dive into the logic behind decision tables and extensive research to determine how to improve it. The limited availability of online support further complicated the process, making my research more time-intensive and difficult than initially expected. However, nce i gained a clearer understanding of how decision tables function and their intended behavior, I were able to restructure our model effectively.

Another major challenge was the lack of a functional online simulator for Camunda DMN, which made testing my solution impossible. This limitation forced to rely on theoretical validations and manual checks.

Additionally, manually entering the numerous rules required to map all possible outputs proved to be both tedious and time-consuming.

Despite these obstacles, i successfully refined my model and ensured it aligned with the intended decision-making framework.

Prolog and Logical Reasoning

Prolog is a declarative programming language well-suited for implementing logical rules and querying knowledge bases. Its emphasis on recursion, pattern matching, and logical inference makes it ideal for defining relationships and constraints in a structured manner. In my project, i implemented rules to map user preferences—such as lactose-free, gluten-free, or low-calorie requirements—to meals and their associated ingredients. This enabled us to efficiently filter and retrieve relevant meals that satisfied multiple constraints simultaneously. Despite the limited built-in support for conventional arithmetic operations, Prolog’s logic-driven approach proved more than sufficient for my needs.

Advantages and Highlights

What I appreciated most about Prolog was its declarative nature, which allowed to define complex relationships between meals and ingredients in a concise and intuitive way. The ability to apply layered constraints easily was particularly valuable when addressing scenarios involving multiple, overlapping dietary restrictions. Prolog’s built-in predicates and custom rules provided a flexible framework for modeling these constraints efficiently, making the logic of the system both clear and manageable.

Additionally, the online tool i used for Prolog was very user-friendly. It supported my process of modeling the solution effectively, providing helpful features that guided through the development and debugging stages. The ease with which i could experiment with different rules and queries in a logical, structured environment contributed significantly to the project’s success.

Challenges and Issues

The main challenge for encountered with Prolog was during the revision phase after first submission. Due to time constraints and the lack of comprehensive online

documentation, we were unable to improve or refine the initial solution i had proposed. As it was first time using Prolog, i lacked the experience and familiarity to navigate its intricacies efficiently. Although i manage to implement a solution, i were unable to optimize it. Despite this, Prolog remained enjoyable throughout the project.

Protégé for Ontology Creation

Protégé is a powerful ontology development tool that allowed to define classes, properties, and relationships within my personalized menu system. Through Protégé, i were able to create a formal representation of the given knowledge domain by modeling entities such as Ingredient, Meal, Menu, and various attributes like calorie levels and dietary restrictions. Using a combination of object properties (e.g., hasMeal, MealHasGluten) and data properties (e.g., hasCalories), i established a structured framework to represent the logic behind the personalization of a menu based on clients preferences.

Advantages and Highlights

What I liked most about Protégé was how the more I learned, the more I realized its full potential. After conducting several tests, I were able to implement complex relationships between classes, which greatly enhanced the depth of my ontology. The graph diagram feature was particularly helpful in visualizing and understanding the hierarchy of the model i created.

Additionally, Protégé's support for OWL (Web Ontology Language) made defining intricate relationships and attributes for each class much easier and more structured. This flexibility allowed to build a robust and detailed ontology that perfectly suited the needs of my project.

Challenges and Issues

A significant challenge i faced was the overall complexity of ontology modeling, especially since this was my first time working on a project of this scale. The learning curve was steep, and it took considerable effort to understand how to effectively model my knowledge base. and also had to conduct extensive research to figure out how to achieve the specific functionalities i desired within Protégé.

In addition, the manual data entry process proved to be another major challenge. Populating the ontology with data by hand was not only time-consuming but also error-prone, particularly given the large volume of information required for meal and ingredient instances. This made the process slower and more tedious than i had initially anticipated.

SHACL Shapes and SWRL Rules for Validation and Constraints

SHACL is a framework designed to validate RDF data based on a set of predefined constraints, or "shapes". These shapes define the structure and relationships that data must adhere to, ensuring that data conforms to the desired specification. In my project, SHACL shapes were used to validate instances of the classes in my ontology, such as ensuring that specific properties like hasMeal or isInMeal were correctly implemented.

This provided us with an efficient way to ensure data consistency and correctness across my personalized menu system.

SWRL, on the other hand, was used for more complex inferences. By using SWRL, we could create rules that infer new relationships based on existing data. These rules allowed us to define logical inferences, such as classifying meals based on their ingredients or calorie content. Following my first submission, we expanded the scope of these rules to enable deeper inference capabilities, enriching the knowledge base and improving the flexibility of the system. This dynamic reasoning was essential for populating the menu objects with meals tailored to specific user preferences or dietary restrictions.

Advantages and Highlights

I found both SHACL shapes and SWRL rules to be extremely helpful in my project. SHACL shapes were invaluable for validating ontology and ensuring that i correctly defined all properties and relationships. This validation process allowed to catch errors early, providing confidence that the ontology was structurally sound and consistent.

On the other hand, SWRL rules significantly enhanced the ability to populate the ontology with inferred data, without the need to manually insert every single data point. This was particularly useful for generating dynamic relationships based on existing data, allowing us to automatically classify meals or ingredients based on certain characteristics (e.g., lactose-free, gluten-free). SWRL enabled to define complex relationships between objects in the ontology and infer new information that wasn't explicitly stated, which greatly improved the flexibility of the system and made the ontology more.

Challenges and Issues

The limitations i encountered with SHACL and SWRL were due to the complexity of applying validation and inference in a dynamic and multi-layered domain like personalized meal recommendations. SHACL is powerful for structural validation but less effective for deep reasoning tasks. On the SWRL side, while it facilitated logical inferences, managing a large set of rules became increasingly challenging. As the number of dietary restrictions and meal preferences grew, maintaining a comprehensive and clear set of SWRL rules required significant effort. Balancing the complexity of the rules with system performance and clarity presented challenges, but overall, both SHACL and SWRL were invaluable for enhancing the flexibility and automation of the system.

SPARQL and Graph DB for Querying and Data Storage

SPARQL is a powerful query language used for retrieving and manipulating data stored in RDF format. It allow to query of ontology and extract specific meals or ingredients that met certain dietary restrictions or preferences by defining patterns and conditions.

Advantages and Highlights

The integration with Graph DB and SPARQL was relatively smooth and straight forward. The familiarity with SQL-like syntax made SPARQL queries easier to handle.

The simplicity of both Graph DB and SPARQL contributed to a smooth experience overall, allowing me to focus more on refining the system rather than spending time troubleshooting the tools and correcting errors.

Challenges and Issues

I didn't encounter any significant problems with Graph DB and SPARQL during the project. However, if I had to point out one issue, it would be the graph visualization on the Graph DB website. It was a bit challenging to navigate, and finding the specific nodes I am looking for wasn't always straightforward. This made it harder to quickly interpret and manipulate the data in the graph interface, but it didn't impact the overall functionality of Graph DB or the ease of querying with SPARQL.

AOAME and Jena Fuseki

Using Jena Fuseki and AOAME, i were able to define and execute a SPARQL query on the Jena Fuseki server to retrieve a personalized menu directly from the ontology. This query was designed to extract meals tailored to the client's preferences based on the extended classes and properties that i had defined within AOAME.

Advantages and Highlights

I found AOAME to be both interesting and highly beneficial due to its ability to seamlessly integrate BPMN diagrams with my ontology. This capability enabled to extend BPMN's classes and properties to better fit the needs of my personalized menu system. After creating the BPMN diagram to describe the process to create a personalized menu, implementing the query on Jena Fuseki and running it became a very straightforward task. The clear instructions and examples allowed us to quickly move from conceptualization to implementation, making the query execution simple and efficient once the initial setup was complete. This greatly reduced the complexity and allowed to focus on refining the logic and tailoring the solution to meet the project's needs. I also found the tutorials provided for Jena Fuseki and AOAME extremely helpful in guiding through the setup and usage process. These tutorials were invaluable in helping to understand how to leverage these tools for my project.

Challenges and Issues

One of the main challenges we faced was ensuring a smooth integration between the BPMN diagram and the extended ontology in AOAME. Mapping BPMN elements to the correct ontology classes and properties required careful consideration, as i needed to extend the default BPMN classes to accommodate specific meal personalization features. This process was further complicated by the need to create custom SPARQL queries in Jena Fuseki that could effectively retrieve data from the newly extended ontology.

Additionally, i encountered many bugs that required multiple uninstalls and reinstalls of the software to resolve. This process was time-consuming and often disrupted progress.

Future Developments

Several improvements could be made to enhance the functionality and complexity of my project. First, expanding the range of user preferences, such as incorporating categories for people with vegan diets or other specific dietary restrictions, would make the system more versatile and inclusive.

Additionally, standardizing the Prolog file would increase its compatibility and maintainability, making it more in line with best practices in Prolog programming.

Furthermore, enhancing the SWRL rules could bring more intricate relationships between data, creating a richer and more dynamic decision-making framework. For example, by modifying the existing SWRL rules for meal categorization, i could enable the ontology to infer more detailed connections, such as showing medium and low level calories meals when selecting the high calorie level.

