University of Camerino

SCHOOL OF SCIENCE AND TECHNOLOGY

Corso di Master Degree in Computer Science (LM-18)

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

Authors
Christian Bonsignore
Alessandro Quercetti

Professors
Knut Hinkelmann
Emanuele Laurenzi

A.Y. 2024/2025

Indice

Elenco dei codici

Elenco delle figure

Elenco delle tabelle

1. Introduction

In recent years, growing awareness around food-related health issues, dietary restrictions, and personal preferences has highlighted the need for intelligent systems capable of supporting informed meal choices. This project addresses this need by exploring a knowledge-based approach to personalized menu generation, where users can receive suggestions for dishes tailored to their nutritional needs, allergies, or ethical choices such as vegetarianism. The central idea is to produce a knowledge base to help represent knowledge about meals, ingredients, and their nutritional properties in a formal, structured way, allowing for automated reasoning and intelligent filtering. Several independent technologies and paradigms are considered — each offering a different solution to the same underlying problem: how to effectively and flexibly filter and recommend meals based on user-defined criteria.
These approaches include the use of ontologies and semantic reasoning (with **OWL** and **SWRL**), logic programming (**PROLOG**), query languages for structured data (**SPARQL**), and agile-inspired modeling strategies. Each component contributes a different perspective on how to handle the complexity of dietary constraints and user preferences, illustrating the versatility of knowledge-based systems in the food domain.

1.1 Project Objective

The main objective of this project is to investigate and model different knowledge-based strategies for building a personalized meal recommendation system. More specifically, the aim is to explore how formal representations of meals, ingredients, and dietary constraints can be used to support intelligent decision-making.
The system must be capable of:

- Representing meals and their properties (such as calorie count, presence of gluten or lactose, vegetarian status) in a machine-readable format.
- Representing meals and their properties (such as calorie count, presence of gluten or lactose, vegetarian status) in a machine-readable format.
- Applying rule-based logic or reasoning to determine whether a meal satisfies specific user-defined filters (e.g. max calories, allergies, dietary requirements).
- Filtering out meals that contain undesired ingredients or conflict with nutritional goals.
- Demonstrating multiple independent techniques—including semantic web technologies, logical reasoning, and declarative querying—to solve the same core problem from different angles.

By comparing and implementing different methodologies, the project aims to evaluate the strengths and limitations of each approach in handling dietary personalization, paving the way for more advanced, flexible, and user-centric food recommendation systems.

1.2 Menu Composition

The menu that we created for our implementation is formed by 27 ingredients and 12 dishes. We represented each ingredient using the same basic properties across the components of the project:

- *name* of the ingredient,
- approximate *calorie count* per serving,
- presence of *gluten*,
- presence of *lactose*,
- compliance to a common *vegetarian* diet.

Below, find two tables explaining our decisions on the menu: the first table (1.1) contains all the dishes with the respective ingredients, the second (1.2) shows the relation between the single ingredients and the respective properties.

Dish	**Ingredients**
tagliereFormaggi	caciotta, pecorino, parmigiano
tagliereSalumi	salame, mortadella, coppa
crescia	acqua, olio, farina, rosmarino
pastaPesto	pasta, olio, basilico, aglio, parmigiano
pastaCarbonara	guanciale, pecorino, uova, pasta, pepe
risottoPomodoro	riso, pomodoro, basilico, parmigiano, burro
pizzaMargherita	farina, olio, acqua, pomodoro, mozzarella, basilico
pizzaTonnoCipolla	farina, olio, acqua, tonno, cipolla
pizzaDiavola	farina, acqua, olio, spianata, pomodoro
insalataLegumi	ceci, lenticchie, fagioli, pepe, olio, cipolla
uovaSode	uova, olio, pepe
tagliataRosmarino	carne, olio, rosmarino, aglio

Tabella 1.1: Dishes and their respective ingredients

Ingredient	**Calorie Count**	**Gluten Free**	**Lactose Free**	**Vegetarian**
acqua	0	✓	✓	✓
aglio	5	✓	✓	✓
basilico	1	✓	✓	✓
burro	102	✓	✗	✓
caciotta	110	✓	✗	✓
carne	250	✓	✓	✗
ceci	180	✓	✓	✓
cipolla	40	✓	✓	✓
coppa	270	✓	✓	✗
fagioli	120	✓	✓	✓
farina	360	✗	✓	✓
guanciale	300	✓	✓	✗
lenticchie	115	✓	✓	✓
mortadella	310	✓	✗	✗
mozzarella	85	✓	✗	✓
olio	120	✓	✓	✓
parmigiano	110	✓	✗	✓
pasta	220	✗	✓	✓
pecorino	130	✓	✓	✓
pepe	2	✓	✓	✓
pomodoro	20	✓	✓	✓
riso	200	✓	✓	✓
rosmarino	3	✓	✓	✓
salame	300	✓	✓	✗
spianata	310	✓	✓	✗
tonno	140	✓	✓	✗
uova	70	✓	✓	✓

Tabella 1.2: Ingredient attributes: nutritional and dietary compatibility

2. Decision Modeler: Trisotech

Trisotech is a cloud-based platform designed to support the modeling, simulation, and management of business and decision processes through internationally recognized standards such as **BPMN** (Business Process Model and Notation), **DMN** (Decision Model and Notation), and **CMMN** (Case Management Model and Notation). Thanks to its intuitive and collaborative interface, Trisotech enables interdisciplinary teams to define and share complex decision logic in a clear and structured manner.
In the context of this project, we specifically used the DMN functionalities offered by Trisotech to formally represent and manage the rules for dish selection and menu composition. This allowed us to model the decision-making process based on nutritional preferences and dietary constraints, facilitating the definition and execution of logic such as filtering by calorie threshold, allergen presence, and dietary requirements (e.g., vegetarian options).
Before going further, it is necessary to dwell on the description of these two concepts:

Decision table : A decision table is a structured way to model and represent decision logic. In DMN, a decision table consists of:

- Inputs (conditions or criteria),
- A set of rules (rows),
- Outputs (results or actions based on the conditions).

Each row represents a possible rule or scenario, and the table as a whole defines how decisions should be made based on various combinations of inputs. Decision tables help to make decision logic clear, auditable, and easy to maintain.

FEEL expression : **FEEL** (Friendly Enough Expression Language) is the standard expression language used in DMN to define conditions, computations, and decision logic. It is designed to be readable by business users while still being precise and formal. FEEL expressions are used inside decision tables to define things like input conditions, output values, and logic for complex calculations or filters.

2.1 Our Decision Models

This section presents the details of our DMN-based implementations within the platform. In an effort to meet the requirements in the best possible way, we produced two different solutions for this task, each one employing a different level of complexity.

2.1.1 First Solution: Straightforward Implementation without FEEL Expressions

The first solution is derived from the need to avoid using high level constructs, as it was said during the lectures. It has some limitations: it can't count the calories nor exclude dishes because of the clients' allergies to specific ingredients.
It consists of two *decisions*, two *knowledge bases*, and four *input data*, as you can see in figure 2.1. Both the decisions are handled by Decision Tables, which we will see later on. We can divide the process in two parts: one about deciding on the ingredients and then

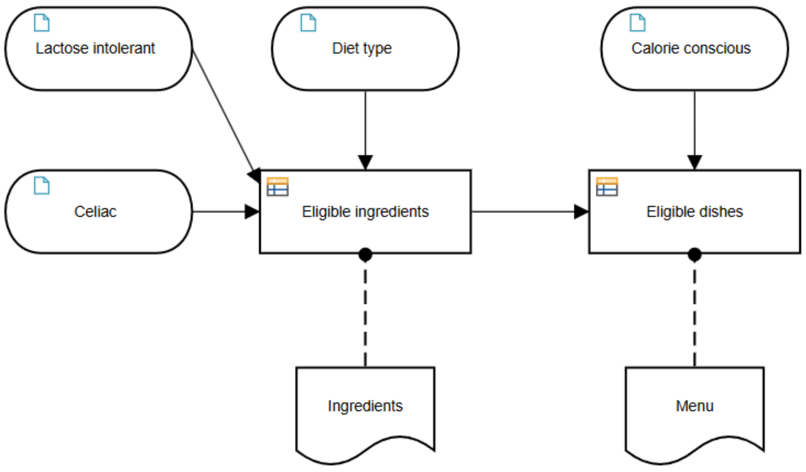
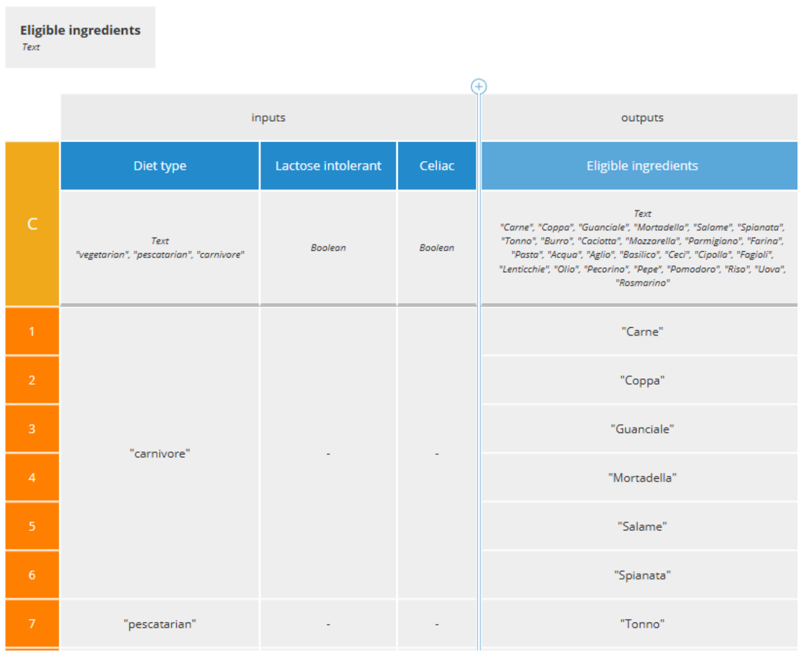


Figura 2.1: Straightforward Implementation without FEEL Expressions

the second on the dishes.
The first decision takes its knowledge from the "Ingredients" knowledge base and requires the user to input if they're lactose intolerant, celiac and their diet type. Being celiac

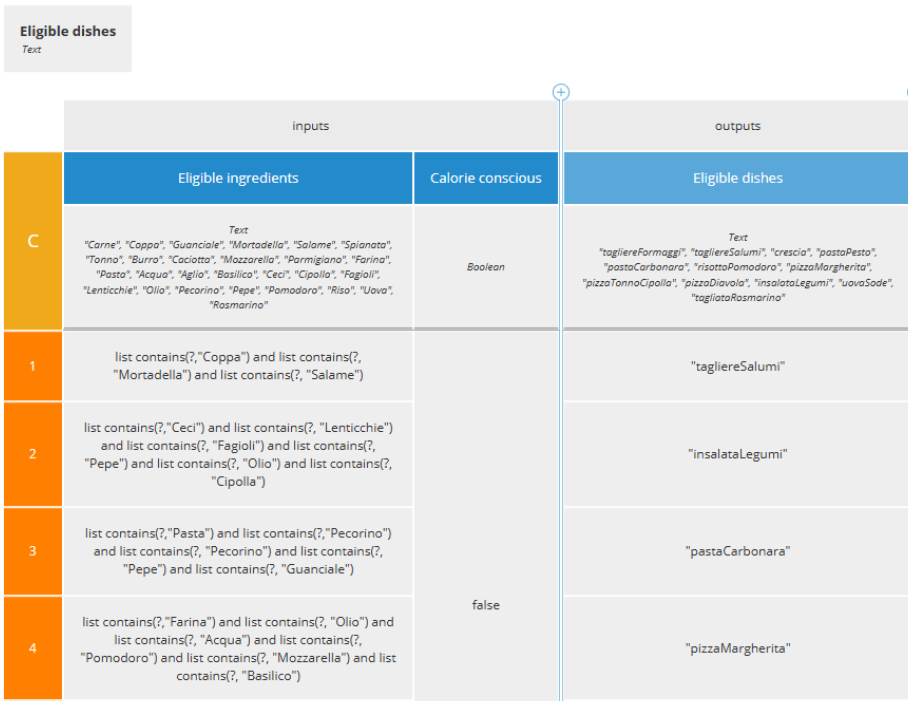


Eligible ingredients
Text

	inputs			outputs
C	Diet type	Lactose intolerant	Celiac	Eligible ingredients
	Text *"vegetarian", "pescatarian", "carnivore"*	*Boolean*	*Boolean*	*Text* *"Carne", "Coppa", "Guanciale", "Mortadella", "Salame", "Spianata", "Tonno", "Burro", "Caciotta", "Mozzarella", "Parmigiano", "Farina", "Pasta", "Acqua", "Aglio", "Basilico", "Ceci", "Cipolla", "Fagioli", "Lenticchie", "Olio", "Pecorino", "Pepe", "Pomodoro", "Riso", "Uova", "Rosmarino"*
1	"carnivore"	-	-	"Carne"
2				"Coppa"
3				"Guanciale"
4				"Mortadella"
5				"Salame"
6				"Spianata"
7	"pescatarian"	-	-	"Tonno"

Figura 2.2: Snapshot of a part of the decision table "Eligible Ingredients"

or lactose intolerant is handled with a boolean value (*true, false*), while the "Diet type" input requires a textual input with a constraint on the possible values (only the strings *"vegetarian"*, *"pescatarian"*, and *"carnivore"* are allowed).
The **Hit Policy** on this table is *COLLECT*, which means that all rules whose conditions are satisfied will be executed, and their outputs will be collected together. This hit policy is useful in this situation because multiple rules may apply at the same time and I want to aggregate their outputs to obtain a list of eligible ingredients, as the name of the output column in figure 2.2 suggests. So the output value of the single fired rule is again a string with the name of the valid ingredient, from the table 1.2.
The second decision is called "Eligible Dishes". It takes new knowledge about the dishes (the list can be found in the table 1.1), and requires two inputs: the output from the "Eligible Ingredients" decision and a new one, "Calorie conscious". This one is represented by a boolean value to help the client discriminate between dishes with a higher or lower calorie count. The chosen threshold is 500 calories.
As it can be seen in figure 2.3, the Hit Policy is "COLLECT" again, because of the need to fire multiple rules corresponding to multiple dishes being eligible for the profile of the client. The output is once again a string with a constraint on the values bound to the


Eligible dishes
Text

	inputs		outputs
C	Eligible ingredients	Calorie conscious	Eligible dishes
	Text *"Carne", "Coppa", "Guanciale", "Mortadella", "Salame", "Spianata", "Tonno", "Burro", "Caciotta", "Mozzarella", "Parmigiano", "Farina", "Pasta", "Acqua", "Aglio", "Basilico", "Ceci", "Cipolla", "Fagioli", "Lenticchie", "Olio", "Pecorino", "Pepe", "Pomodoro", "Riso", "Uova", "Rosmarino"*	*Boolean*	*Text* *"taglíereFormaggi", "taglíereSalumi", "crescia", "pastaPesto", "pastaCarbonara", "risottoPomodoro", "pizzaMargherita", "pizzaTonnoCipolla", "pizzaDiavola", "insalataLegumi", "uovaSode", "tagliataRosmarino"*
1	list contains(?,"Coppa") and list contains(?, "Mortadella") and list contains(?, "Salame")	false	"taglièreSalumi"
2	list contains(?,"Ceci") and list contains(?, "Lenticchie") and list contains(?, "Fagioli") and list contains(?, "Pepe") and list contains(?, "Olio") and list contains(?, "Cipolla")		"insalataLegumi"
3	list contains(?,"Pasta") and list contains(?,"Pecorino") and list contains(?, "Pecorino") and list contains(?, "Pepe") and list contains(?, "Guanciale")		"pastaCarbonara"
4	list contains(?,"Farina") and list contains(?, "Olio") and list contains(?, "Acqua") and list contains(?, "Pomodoro") and list contains(?, "Mozzarella") and list contains(?, "Basilico")		"pizzaMargherita"

Figura 2.3: Snapshot of a part of the decision table "Eligible Dishes"

names of the dishes in the menu.
The output coming from the previous decision table is a string of ingredients that are eligible based on the client's profile, that means that if all the ingredients contained in a dish can be found in that list, the dish is eligible. To achieve the wanted result, I used the method `list.contains()` in the row for as many times as the number of ingredients contained in the respective dish.

2.1.2 Second Solution: Complex Implementation employing FEEL Expressions

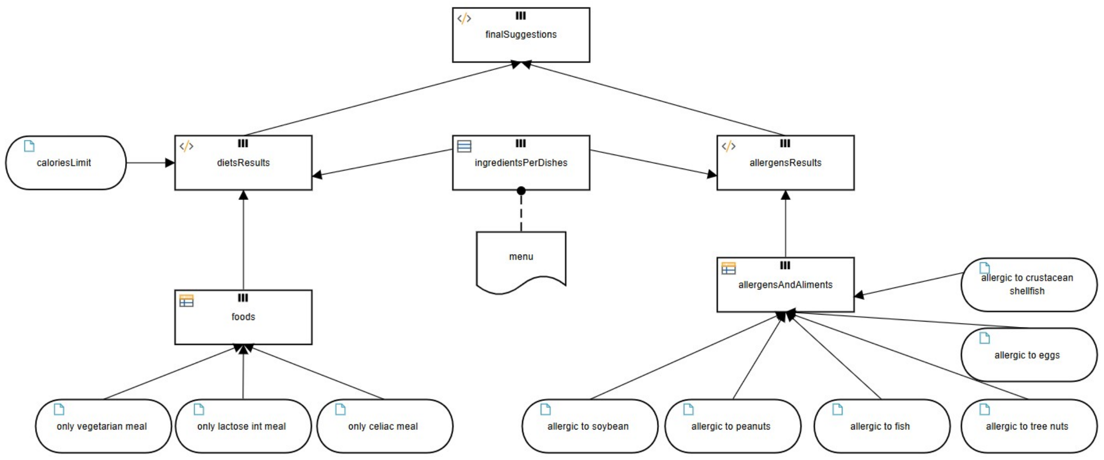
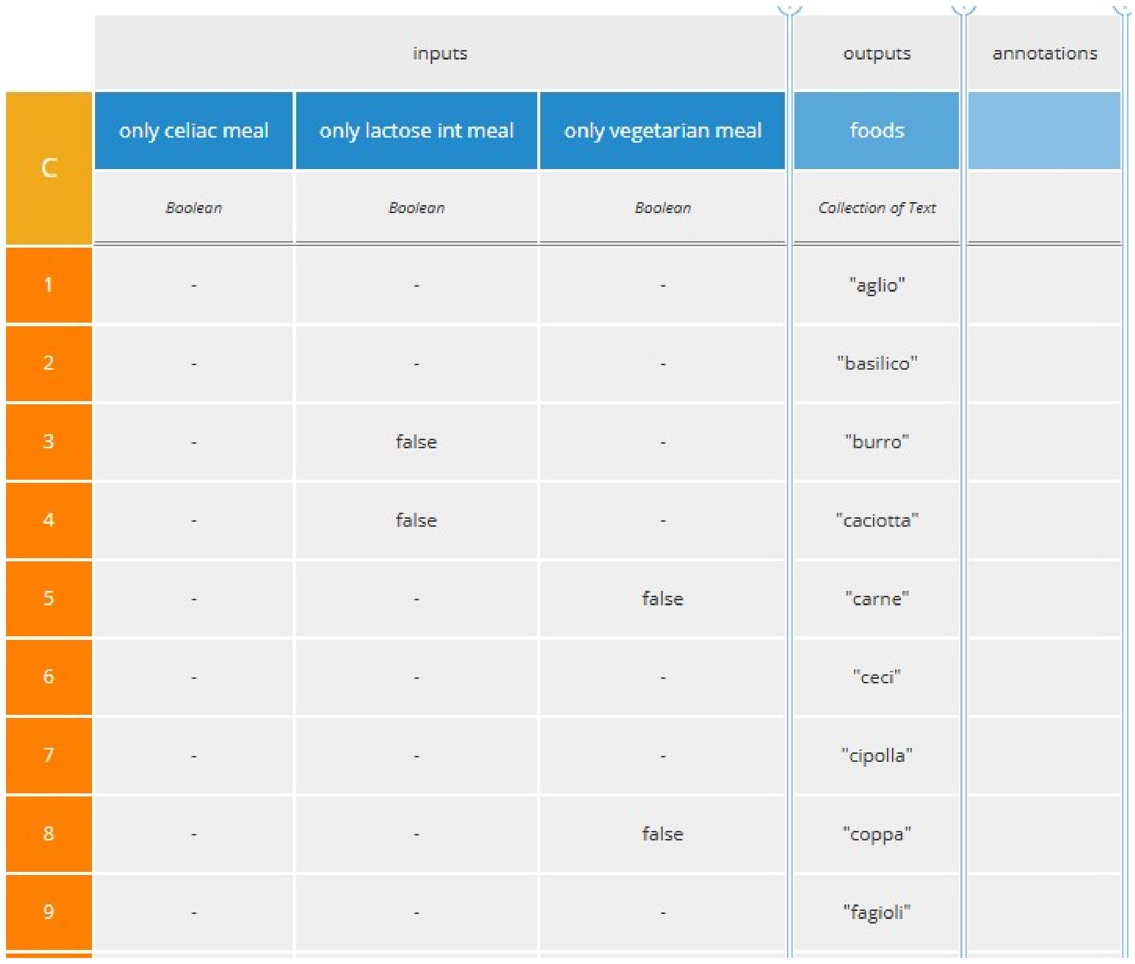


Figura 2.4: Complex DMN employing FEEL Expressions

As we can see in the image 2.4, the process starts with inputs of diet type and allergens (each is a Boolean value, true or false):

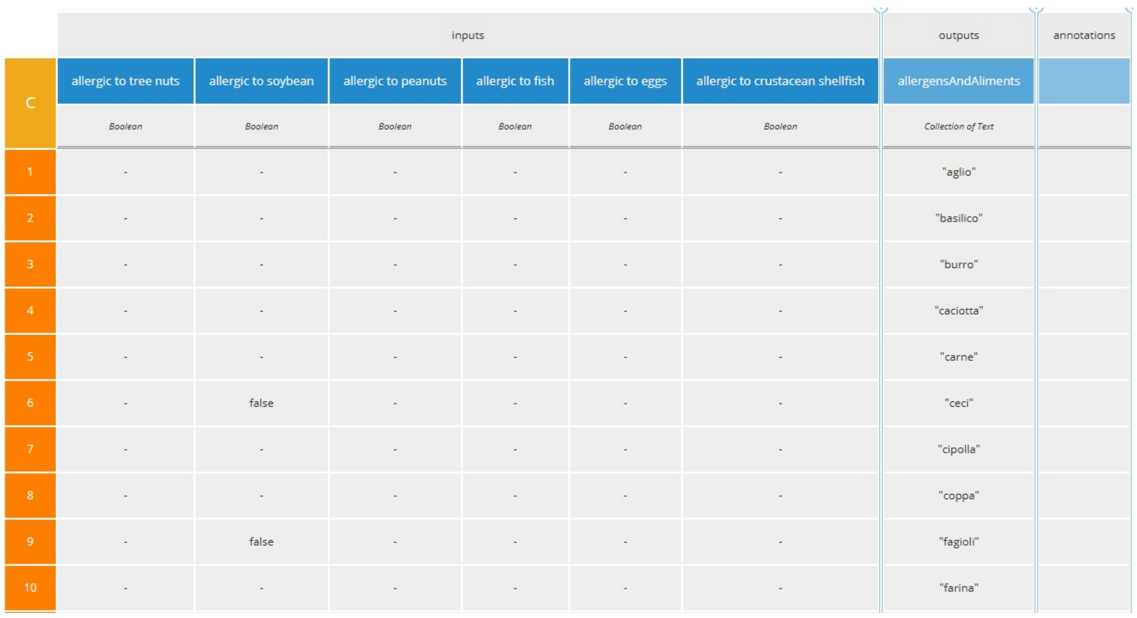
- diet inputs: only vegetarian meal, only lactose intolerant meal, only celiac meal
- allergies: soybean, peanuts, fish, tree nuts, eggs, crustacean shellfish

There is also a third input that represents the calories limit for a single dish; we opted for this implementation to make the solution more streamlined and direct.
The menu is represented by the list called *ingredientPerDishes*, where we can find the dish with the corrisponding ingredients and the calories value. *Example: ["taglièreFormaggi", ["caciotta", "pecorino", "parmigiano"], 350]*
Then we can find two decision tables, *food* and *allergensAndAliments*, that provide two lists of ingredients filtered using the information provided by the user. The images 2.5 and 2.6 are part of these tables.



	inputs			outputs	annotations
C	only celiac meal	only lactose int meal	only vegetarian meal	foods	
	Boolean	*Boolean*	*Boolean*	*Collection of Text*	
1	-	-	-	"aglio"	
2	-	-	-	"basilico"	
3	-	false	-	"burro"	
4	-	false	-	"caciotta"	
5	-	-	false	"carne"	
6	-	-	-	"ceci"	
7	-	-	-	"cipolla"	
8	-	-	false	"coppa"	
9	-	-	-	"fagioli"	

Figura 2.5: Decision table about the type of diet



	inputs						outputs	annotations
C	allergic to tree nuts	allergic to soybean	allergic to peanuts	allergic to fish	allergic to eggs	allergic to crustacean shellfish	allergensAndAliments	
	Boolean	*Boolean*	*Boolean*	*Boolean*	*Boolean*	*Boolean*	*Collection of Text*	
1	-	-	-	-	-	-	"aglio"	
2	-	-	-	-	-	-	"basilico"	
3	-	-	-	-	-	-	"burro"	
4	-	-	-	-	-	-	"caciotta"	
5	-	-	-	-	-	-	"carne"	
6	-	false	-	-	-	-	"ceci"	
7	-	-	-	-	-	-	"cipolla"	
8	-	-	-	-	-	-	"coppa"	
9	-	false	-	-	-	-	"fagioli"	
10	-	-	-	-	-	-	"farina"	

Figura 2.6: Decision table about allergens

The results of these decision table are filtered using the two FEEL expressions *dietResult* and *allergensResults*, and finally combined in *finalSuggestions*.

Codice 2.1: FEEL expression dietsResults

```
for item in ingredientsPerDishes
    return if (every ingr in item[2] satisfies ingr in foods)
        and (item[3] <= caloriesLimit)
    then item[1]
    else "EXCLUDE"
```

Codice 2.2: FEEL expression allergensResult

```
for item in ingredientsPerDishes
    return if every ingr in item[2] satisfies ingr in
    allergensAndAliments then item[1] else "EXCLUDE"
```

Codice 2.3: FEEL expression finalSuggestions

```
for x in allergensResults return
  if x in dietsResults then x else "EXCLUDE"
```

At the end of the computation, we obtain a list of dishes consisting of actual names and the term \EXCLUDE". Since Trisotech does not support removing elements from a list, we use this term to indicate the dishes that do not meet the client's requirements.
Below we find an example of computation, from the insertion of the inputs to the final output with the suggested dishes:

1. - only vegetarian meal: true
   - only lactose intolerant meal: false
   - only celiac meal: false
   - allergic to soybean: false
   - allergic to peanuts: false
   - allergic to fish : false
   - allergic to tree nuts: false
   - allergic to eggs: true
   - allergic to crustacean shellfish : false
   - caloriesLimit: 500
2. result list from *foods*: aglio, basilico, burro, caciotta, ceci, cipolla, fagioli, farina, lenticchie, mozzarella, olio, parmigiano, pasta, pecorino, pepe, pomodoro, riso, rosmarino, uova
3. result list from *allergensAndAliments*:aglio, basilico burro caciotta, carne, ceci, cipolla, coppa, fagioli, farina, guanciale, lenticchie, mortadella, mozzarella, olio, parmigiano, pasta, pecorino, pepe, pomodoro, riso, rosmarino, salame, spianata, tonno
4. result list from *dietsResults*: tagliereFormaggi, EXCLUDE, EXCLUDE, pastaPesto, EXCLUDE, risottoPomodoro, EXCLUDE, EXCLUDE, EXCLUDE, EXCLUDE, uovaSode, EXCLUDE

5. result list from *allergensResults*: tagliereFormaggi, tagliereSalumi, EXCLUDE, pastaPesto, EXCLUDE, risottoPomodoro, EXCLUDE, EXCLUDE, EXCLUDE, insalataLegumi, EXCLUDE, tagliataRosmarino

6. result list from *finalSuggestions*: tagliereFormaggi, EXCLUDE, EXCLUDE, pastaPesto, EXCLUDE, risottoPomodoro, EXCLUDE, EXCLUDE, EXCLUDE, EXCLUDE, EXCLUDE, EXCLUDE.

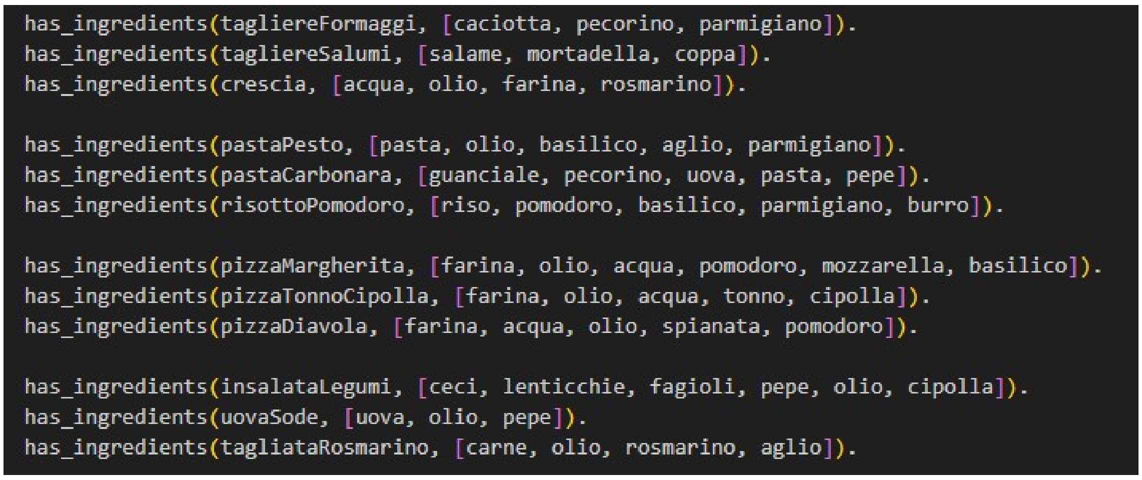
3. PROLOG

PROLOG is a logic programming language widely used in fields such as artificial intelligence, natural language processing, and knowledge representation. Its declarative paradigm allows developers to define a set of facts and rules, letting the language engine infer solutions through logical reasoning rather than explicit sequences of instructions.
The core idea behind PROLOG is to describe relationships between entities and let the system deduce answers by querying these relationships. This makes it particularly well-suited for applications involving rule-based decision-making, constraint satisfaction, and symbolic reasoning.
In the context of our project, PROLOG was employed to model and reason over the data that we have seen in the previous chapters. The following section presents the details of our implementation.

3.1 Menu data in PROLOG

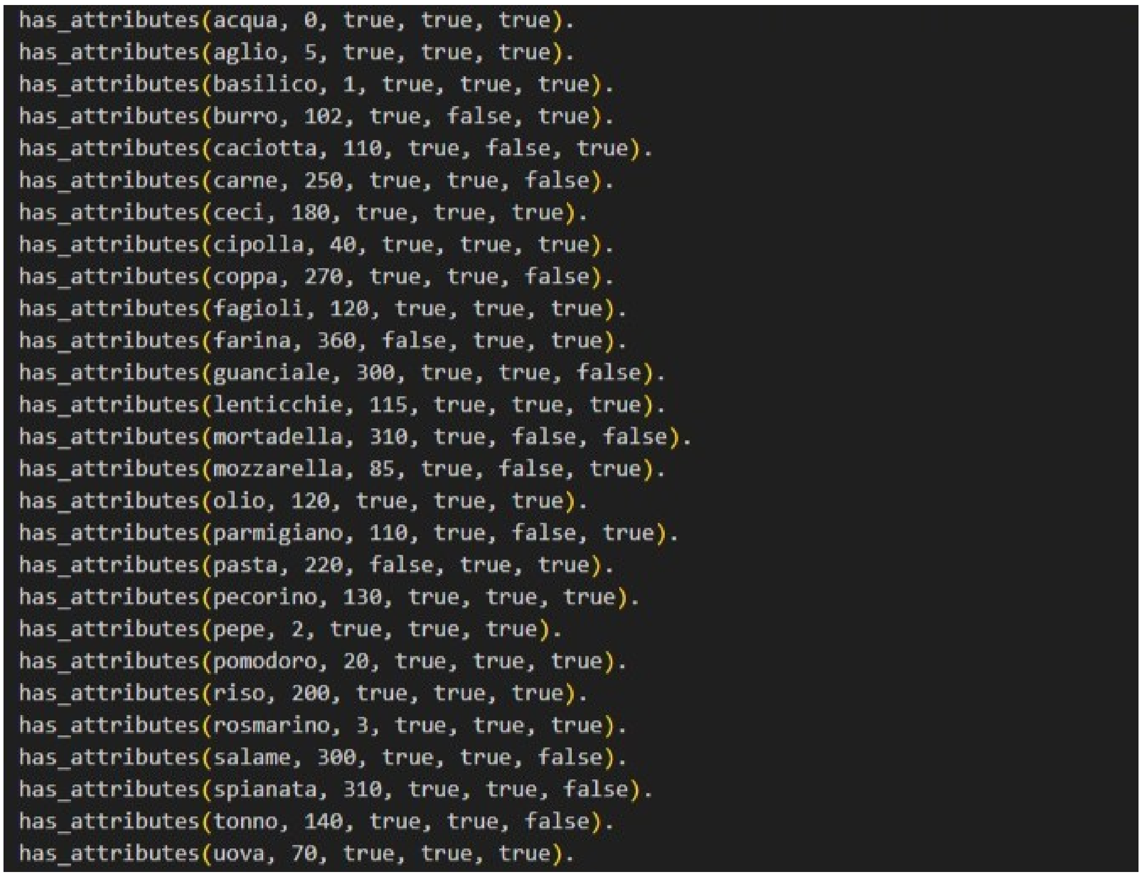
The following images represent the implementation in PROLOG of the menu:

- fig 3.1 is the list of the dishes with their ingredients
- fig 3.2 is the list of the ingredients with their attributes, following the schema: *hasAttributes(Ingredient, CalorieCount, isGlutenFree, isLactoseFree, isVegetarian)*



```
has_ingredients(taglierеFormaggi, [caciotta, pecorino, parmigiano]).
has_ingredients(taglierеSalumi, [salame, mortadella, coppa]).
has_ingredients(crescia, [acqua, olio, farina, rosmarino]).

has_ingredients(pastaPesto, [pasta, olio, basilico, aglio, parmigiano]).
has_ingredients(pastaCarbonara, [guanciale, pecorino, uova, pasta, pepe]).
has_ingredients(risottoPomodoro, [riso, pomodoro, basilico, parmigiano, burro]).

has_ingredients(pizzaMargherita, [farina, olio, acqua, pomodoro, mozzarella, basilico]).
has_ingredients(pizzaTonnoCipolla, [farina, olio, acqua, tonno, cipolla]).
has_ingredients(pizzaDiavola, [farina, acqua, olio, spianata, pomodoro]).

has_ingredients(insalataLegumi, [ceci, lenticchie, fagioli, pepe, olio, cipolla]).
has_ingredients(uovaSode, [uova, olio, pepe]).
has_ingredients(tagliataRosmarino, [carne, olio, rosmarino, aglio]).
```

Figura 3.1: Dishes with respective ingredients in PROLOG



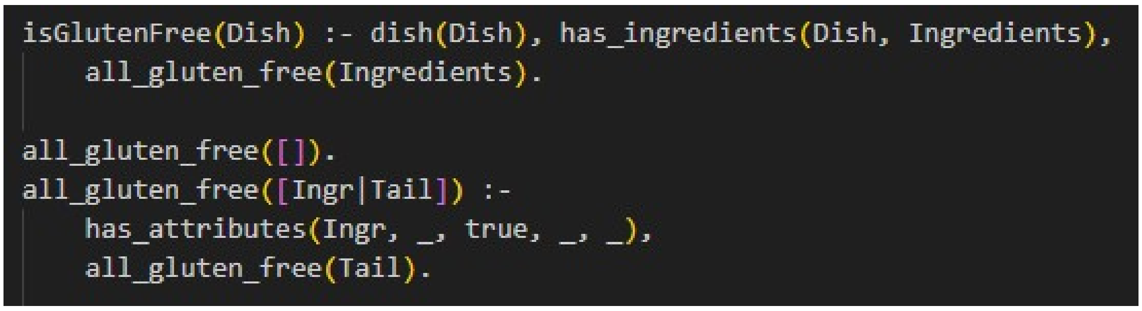
```
has_attributes(acqua, 0, true, true, true).
has_attributes(aglio, 5, true, true, true).
has_attributes(basilico, 1, true, true, true).
has_attributes(burro, 102, true, false, true).
has_attributes(caciotta, 110, true, false, true).
has_attributes(carne, 250, true, true, false).
has_attributes(ceci, 180, true, true, true).
has_attributes(cipolla, 40, true, true, true).
has_attributes(coppa, 270, true, true, false).
has_attributes(fagioli, 120, true, true, true).
has_attributes(farina, 360, false, true, true).
has_attributes(guanciale, 300, true, true, false).
has_attributes(lenticchie, 115, true, true, true).
has_attributes(mortadella, 310, true, false, false).
has_attributes(mozzarella, 85, true, false, true).
has_attributes(olio, 120, true, true, true).
has_attributes(parmigiano, 110, true, false, true).
has_attributes(pasta, 220, false, true, true).
has_attributes(pecorino, 130, true, true, true).
has_attributes(pepe, 2, true, true, true).
has_attributes(pomodoro, 20, true, true, true).
has_attributes(riso, 200, true, true, true).
has_attributes(rosmarino, 3, true, true, true).
has_attributes(salame, 300, true, true, false).
has_attributes(spianata, 310, true, true, false).
has_attributes(tonno, 140, true, true, false).
has_attributes(uova, 70, true, true, true).
```

Figura 3.2: Ingredients and their attributes in PROLOG

3.2 PROLOG rules

Below there is the list of the rules that we used in our implementation:

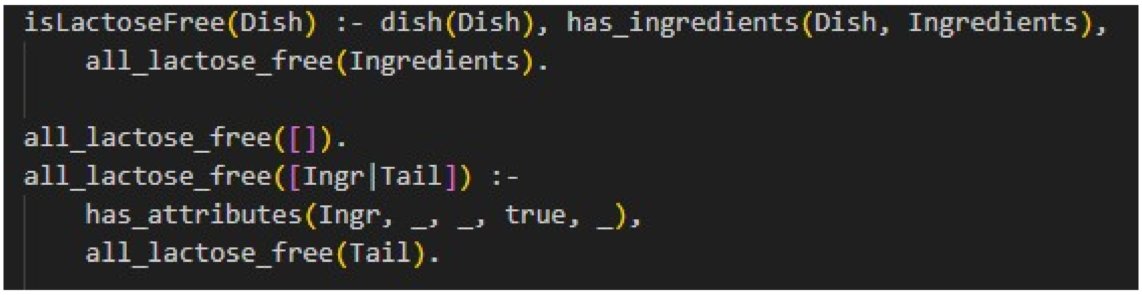
The image 3.3 contains the rule that checks if a dish contains only ingredients without gluten. The main rule, *isGlutenFree(Dish)*, succeeds if the dish has a list of ingredients and all of them are gluten-free. The helper predicate *all_gluten_free* checks that each ingredient in the list has the gluten-free attribute set to true.



```
isGlutenFree(Dish) :- dish(Dish), has_ingredients(Dish, Ingredients),
    all_gluten_free(Ingredients).

all_gluten_free([]).
all_gluten_free([Ingr|Tail]) :-
    has_attributes(Ingr, _, true, _, _),
    all_gluten_free(Tail).
```

Figura 3.3: Rule for determining if a dish is gluten-free

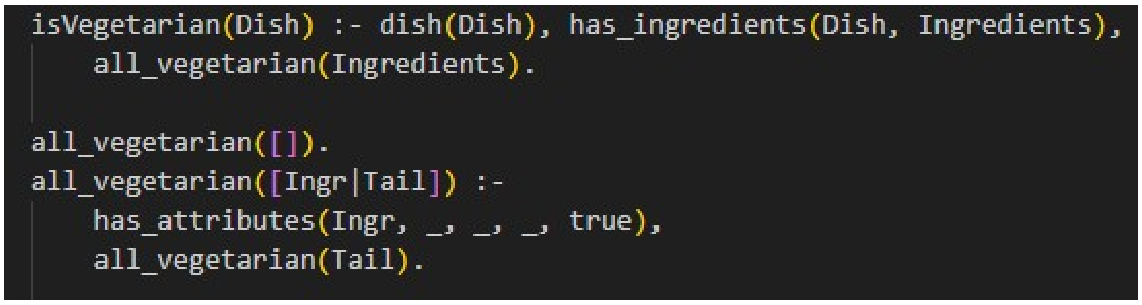
The image 3.4 contains the rule that checks if a dish contains only ingredients without lactose.



```
isLactoseFree(Dish) :- dish(Dish), has_ingredients(Dish, Ingredients),
    all_lactose_free(Ingredients).

all_lactose_free([]).
all_lactose_free([Ingr|Tail]) :-
    has_attributes(Ingr, _, _, true, _),
    all_lactose_free(Tail).
```

Figura 3.4: Rule for determining if a dish is lactose

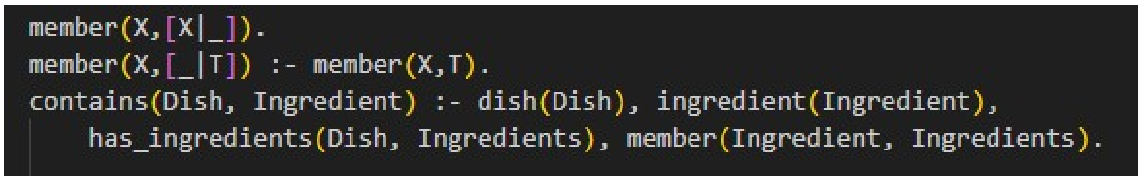
The image 3.5 contains the rule that checks if a dish contains only vegetarian ingredients.



```
isVegetarian(Dish) :- dish(Dish), has_ingredients(Dish, Ingredients),
    all_vegetarian(Ingredients).

all_vegetarian([]).
all_vegetarian([Ingr|Tail]) :-
    has_attributes(Ingr, _, _, _, true),
    all_vegetarian(Tail).
```

Figura 3.5: Rule for determining if a dish is vegetarian

The image 3.6 contains the rule for determining whether a specific ingredient is part of a given dish. The first two clauses define the member/2 predicate, which checks whether an element X belongs to a list. This is a standard list membership definition in PROLOG. The *contains(Dish, Ingredient)* rule uses the member predicate to verify if the Ingredient is present in the list of ingredients associated with the Dish (retrieved through *has_ingredients*).

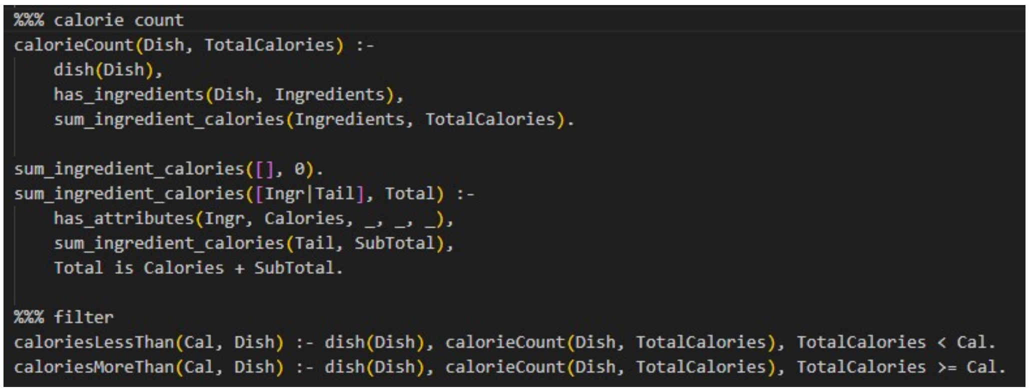


```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
contains(Dish, Ingredient) :- dish(Dish), ingredient(Ingredient),
    has_ingredients(Dish, Ingredients), member(Ingredient, Ingredients).
```

Figura 3.6: Rule for determining if a dish contains a certain ingredient

The image 3.7 contains the rule for calculating the calories in a plate and provides filtering rules based on a calorie threshold. *calorieCount(Dish, TotalCalories)* computes the total number of calories in a dish by summing the calories of all its ingredients. *sum_ingredient_calories(List, Total)*is a helper predicate that recursively sums the calories of each ingredient in the list. *caloriesLessThan(Cal, Dish)* succeeds if the total calories of the given dish are less than Cal, while *caloriesMoreThan(Cal, Dish)* succeeds if the total calories of the given dish are greater than or equal to Cal. The image 3.8 contains the general rule that puts together all the other rules, adding also the filter for the allergic ingredients. As you can see in the image, the rule accepts six parameters:

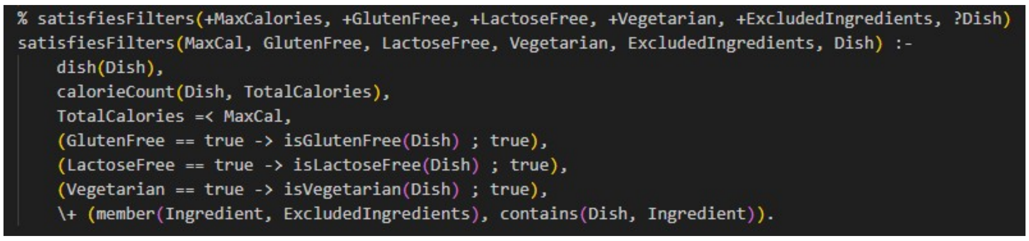
- *MaxCal* is an integer for the maximum number of calories wanted by the client;



```
%%% calorie count
calorieCount(Dish, TotalCalories) :-
    dish(Dish),
    has_ingredients(Dish, Ingredients),
    sum_ingredient_calories(Ingredients, TotalCalories).

sum_ingredient_calories([], 0).
sum_ingredient_calories([Ingr|Tail], Total) :-
    has_attributes(Ingr, Calories, _, _, _),
    sum_ingredient_calories(Tail, SubTotal),
    Total is Calories + SubTotal.

%%% filter
caloriesLessThan(Cal, Dish) :- dish(Dish), calorieCount(Dish, TotalCalories), TotalCalories < Cal.
caloriesMoreThan(Cal, Dish) :- dish(Dish), calorieCount(Dish, TotalCalories), TotalCalories >= Cal.
```

Figura 3.7: Rule for calculating the calories of a dish

- *GlutenFree*, *LactoseFree* and *Vegetarian* are boolean values where *true* means that the client satisfies the condition;
- *ExcludedIngredients* is a list of strings that the client excludes because of allergies or personal preference;
- *Dish* is going to be used as a variable in the query to obtain the list of dishes in output.

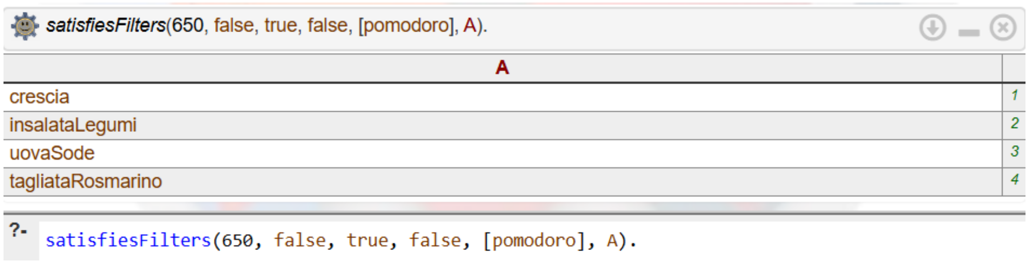


```
% satisfiesFilters(+MaxCalories, +GlutenFree, +LactoseFree, +Vegetarian, +ExcludedIngredients, ?Dish)
satisfiesFilters(MaxCal, GlutenFree, LactoseFree, Vegetarian, ExcludedIngredients, Dish) :-
    dish(Dish),
    calorieCount(Dish, TotalCalories),
    TotalCalories =< MaxCal,
    (GlutenFree == true -> isGlutenFree(Dish) ; true),
    (LactoseFree == true -> isLactoseFree(Dish) ; true),
    (Vegetarian == true -> isVegetarian(Dish) ; true),
    \+ (member(Ingredient, ExcludedIngredients), contains(Dish, Ingredient)).
```

Figura 3.8: General rule that puts together all the other rules

3.3 Decision simulation

The simulation considers a customer who needs to eat foods that are free of lactose, but doesn't necessarily want vegetarian-only dishes. They also dislike the taste of tomato, so they're excluding it. The maximum amount of accepted calories is 650. The returned list contains the meals that meet these preferences.



```
satisfiesFilters(650, false, true, false, [pomodoro], A).
```

A	
crescia	1
insalataLegumi	2
uovaSode	3
tagliataRosmarino	4

```
?- satisfiesFilters(650, false, true, false, [pomodoro], A).
```

Figura 3.9: PROLOG simulation

4. Ontology Engineering

In computer science and knowledge representation, an ontology is a formal specification of a set of concepts and relationships within a particular domain. It provides a shared vocabulary that enables data integration, reasoning, and semantic interoperability. Ontologies are especially useful in fields such as artificial intelligence, the semantic web, and knowledge-based systems, where structured, machine-understandable knowledge is required.
Turtle (Terse RDF Triple Language) is a serialization format for RDF (Resource Description Framework) data. It is used to write RDF graphs in a human-readable way. Turtle files (with `.ttl` extension) represent data as subject-predicate-object triples and are commonly used to define ontologies and linked data. They support namespaces and prefixes, making them concise and readable for developers working with semantic technologies.
Protégé [4] is a free, open-source ontology editor developed by Stanford University. It allows users to create, edit, visualize, and manage ontologies in various formats such as OWL (Web Ontology Language). With a graphical interface and support for reasoning engines, Protégé is widely used in academia and industry for designing ontologies and testing semantic rules like SWRL and SPARQL queries.
This chapter contains the description of our implementation using these technologies.

4.1 Our model

Before moving on, it is necessary to list all the prefixes (shorthand representations for URIs) that we used to organize and simplify the creation of the model and the usage of the functionalities of the next sections.

Codice 4.1: Prefixes for the ontology

```
@prefix : <http://www.semanticweb.org/alessandro/
    ontologies/2025/5/ontologia_ricette#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix swrlb: <http://www.w3.org/2003/11/swrlb#> .
```

In the same order as the previous list, here is the description of the prefixes:

- Our custom ontology namespace, used for defining classes and properties.
- OWL (Web Ontology Language) vocabulary, used for defining ontologies.
- RDF (Resource Description Framework) basics, like rdf:type.

- XML namespace, rarely used directly in OWL but included for completeness.
- XML Schema datatypes, like xsd:string, xsd:integer.
- RDF Schema, used for things like rdfs:subClassOf and rdfs:label.
- SWRL built-in functions, like swrlb:greaterThan, swrlb:add.

As you can see in the image 4.1, we divided the entire data into two main classes, Dish and Ingredients.

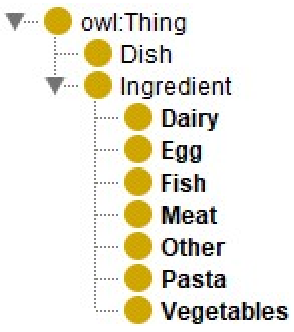


Figura 4.1: Classes in Protege

The following list indicates how the Ingredients are in turn divided into the following classes, which indicate their type:

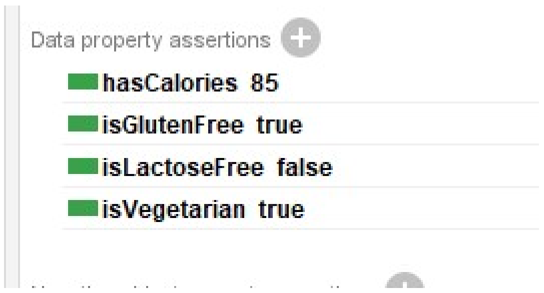
- Diary: burro, caciotta, mozzarella, parmigiano, pecorino
- Egg: uova
- Fish: tonno
- Meat: carne, coppa, guanciale, mortadella, salame, spianata
- Other: acqua, olio
- Pasta: farina, pasta, riso
- Vegetables: aglio, basilico, ceci, cipolla, fagioli, lenticchie, pepe, pomodoro, rosmarino

In the next two figures 4.3 and 4.2 there are examples of data and object properties.

For the ingredients, you can find the following properties:

- *hasCalories*: indicates its amount of calories. The range is *xsd:integer.*
- *isGlutenFree*: the range is *xsd:boolean*, so it could be True or False. Indicates if the ingredient is gluten free (so it does not contain gluten).
- *isLactoseFree*: the range is *xsd:boolean*, indicates if the ingredient is lactose free (so it does not contain lactose).
- *isVegetarian*: the range is *xsd:boolean*, indicates if the ingredient is vegetarian.

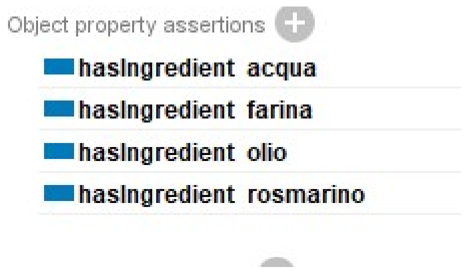
For the dishes there is only one property, *hasIngredient*, which is used to link dishes and ingredients. The domain is *:Dish* and the range is *:Ingredient*.



Data property assertions

- hasCalories 85
- isGlutenFree true
- isLactoseFree false
- isVegetarian true

Figura 4.2: Example of data properties



Object property assertions

- hasIngredient acqua
- hasIngredient farina
- hasIngredient olio
- hasIngredient rosmarino

Figura 4.3: Example of object properties

4.2 SPARQL Queries

SPARQL (SPARQL Protocol and RDF Query Language) [6] is the standard query language used to retrieve and manipulate data stored in the RDF (Resource Description Framework) format, which is commonly used in ontologies. Similar to how SQL works with relational databases, SPARQL allows users to extract structured information from semantic graphs using subject–predicate–object patterns. SPARQL queries can filter results, apply logical conditions, perform aggregations, and navigate relationships defined in the ontology. It is a fundamental tool for exploring, analyzing, and integrating semantic knowledge in the context of ontologies and the Semantic Web.
In the context of our project, we thought of implementing the following features via queries:

- **Vegetarian dishes**

  Codice 4.2: Sparql query for vegetarian dishes

  ```
  SELECT ?dish
  WHERE {
    ?dish a :Dish .
    FILTER NOT EXISTS {
      ?dish :hasIngredient ?i .
      ?i :isVegetarian false .
    }
  }
  ```

  This query retrieves all the dishes that are entirely vegetarian. At the beginning the pattern matches all individuals that are of type *:Dish*, then the filter excludes any dish that contains at least one ingredient (?i) which is not vegetarian (i.e., where *:isVegetarian* is false). In other words, only dishes composed entirely of vegetarian ingredients will be returned.

- **Gluten free dishes**

Codice 4.3: Sparql query for gluten free dishes

```
SELECT ?dish
WHERE {
  ?dish a :Dish .
  FILTER NOT EXISTS {
    ?dish :hasIngredient ?i .
    ?i :isGlutenFree false .
  }
}
```

This query works like the previous one, but checks the *isGlutenFree* property instead of *isVegetarian*.

- **Lactose free dishes**

Codice 4.4: Sparql query for lactose free dishes

```
SELECT ?dish
WHERE {
  ?dish a :Dish .
  FILTER NOT EXISTS {
    ?dish :hasIngredient ?i .
    ?i :isLactoseFree false .
  }
}
```

This one is similar to the others too, but checks the *isLactoseFree* property.

- **Total calories of any dish**

Codice 4.5: Sparql query for total calories

```
SELECT ?dish (STR(SUM(?cal)) AS ?totalCalories)
WHERE {
  ?dish a :Dish ;
        :hasIngredient ?ingredient .
  ?ingredient :hasCalories ?cal .
}
GROUP BY ?dish
```

This query calculates the total number of calories for each dish by summing the calories of all its ingredients. At the beginning, it calculates the sum of the calorie values *(?cal)* for all ingredients of a given dish and converts the result to a string. The resulting value is labeled as *?totalCalories*. Then, for each dish, it retrieves the associated ingredients *(?ingredient)* and gets the calorie value *(?cal)* for each one via the *:hasCalories* property. Finally, it groups the results by dish so that the SUM function aggregates the calories per dish.

- **Dishes below a certain calorie threshold**

  Codice 4.6: Sparql query for calorie threshold

  ```
  SELECT ?dish (STR(SUM(?cal)) AS ?totalCalories)
  WHERE {
    ?dish a :Dish ;
          :hasIngredient ?ingredient .
    ?ingredient :hasCalories ?cal .
  }
  GROUP BY ?dish
  HAVING (SUM(?cal) < 500)
  ```

  This one is an extension of the previous one. It also adds a threshold that limits the calorie value for the output dishes. In this example, this value was set to 500.

- **Filter for multiple conditions**

  Codice 4.7: Sparql query for multiple conditions

  ```
  SELECT DISTINCT ?dish WHERE {
  # Bind all input parameters
  BIND(5000 AS ?maxCalories)
  BIND(false AS ?glutenFree)
  BIND(false AS ?lactoseFree)
  BIND(false AS ?vegetarian)

  # Define excluded ingredients as VALUES
  VALUES ?excludedIng {
  :null
  }

  ?dish a :Dish .

  # Calculate total calories
  {
      SELECT ?dish (SUM(?ingCalories)
              AS ?totalCalories) WHERE {
          ?dish :hasIngredient ?ingredient .
          ?ingredient :hasCalories ?ingCalories .
      }
      GROUP BY ?dish
  }
  FILTER (?totalCalories <= ?maxCalories)

  # Dietary restrictions
  FILTER (IF(?glutenFree, NOT EXISTS { ?dish
      :hasIngredient [ :isGlutenFree false ] }
          , true))
  FILTER (IF(?lactoseFree, NOT EXISTS { ?dish
      :hasIngredient [ :isLactoseFree false ] }
          , true))
  FILTER (IF(?vegetarian, NOT EXISTS { ?dish
  ```

```
        :hasIngredient [ :isVegetarian false ] }
            , true))

    # Excluded ingredients check
    FILTER NOT EXISTS {
        ?dish :hasIngredient ?excludedIng .
    }
    }
```

The last one is very general, it puts together all the previous functionalities and adds something more. This query selects all dishes that:

- Have total calories below a given limit
- Meet dietary preferences (gluten-free, lactose-free, vegetarian if specified)
- Do not contain any explicitly excluded ingredients to check also the allergies

4.3 SHACL Shapes

SHACL (Shapes Constraint Language) [5] is a W3C standard used to validate RDF graphs against a set of constraints or conditions defined in "shapes". These shapes describe the expected structure and content of data, allowing developers to ensure that their ontology-based models remain consistent and semantically correct. In our project, SHACL shapes are used to enforce constraints on data such as ingredient properties or dish composition, serving as a validation layer to support data integrity and reliability in knowledge-based systems.
In particular, we decided to implement these shapes:

- Validates that all dishes have at least one ingredient

  Codice 4.8: MinimumIngredientRequirementShape

```
:MinimumIngredientRequirementShape
    a sh:NodeShape ;
    sh:targetClass :Dish ;
    sh:property [
        sh:path :hasIngredient ;
        sh:minCount 1 ;
        sh:message "A dish must contain at least one ingredient" ;
    ] .
```

  This shape ensures that every dish instance must be associated with at least one ingredient via the *:hasIngredient* property. If a dish does not meet this requirement, a validation message is triggered: *"A dish must contain at least one ingredient"*. This shape supports data integrity by preventing the definition of incomplete or invalid dish entries.

- Validates that all ingredients have properly formatted data properties

  Codice 4.9: IngredientDataQualityShape

```
:IngredientDataQualityShape
    a sh:NodeShape ;
    sh:targetClass :Ingredient ;
    sh:property [
        sh:path :hasCalories ;
        sh:datatype xsd:integer ;
        sh:minInclusive 0 ;
        sh:message "Calories must be a non-negative integer" ;
    ] ;
    sh:property [
        sh:path :isGlutenFree ;
        sh:datatype xsd:boolean ;
        sh:message "Gluten-free flag must be a boolean value" ;
    ] ;
    sh:property [
        sh:path :isLactoseFree ;
        sh:datatype xsd:boolean ;
        sh:message "Lactose-free flag must be a boolean value" ;
    ] ;
    sh:property [
        sh:path :isVegetarian ;
        sh:datatype xsd:boolean ;
        sh:message "Vegetarian flag must be a boolean value" ;
    ] .
```

The *:IngredientDataQualityShape* validates the data quality of instances of the class *:Ingredient*. It ensures that each ingredient has: a *:hasCalories* property with a non-negative integer value *(xsd:integer not less than 0)*, a *:isGlutenFree* flag that must be a boolean (xsd:boolean), a *:isLactoseFree* flag that must be a boolean *(xsd:boolean), a :isVegetarian* flag that must be a boolean *(xsd:boolean)*. Each constraint is accompanied by a message to provide informative feedback in case of validation errors.

- Ensures all ingredients are properly classified into subclasses

Codice 4.10: IngredientClassificationShape

```
:IngredientClassificationShape
    a sh:NodeShape ;
    sh:targetClass :Ingredient ;
    sh:or (
        [ sh:class :Dairy ]
        [ sh:class :Egg ]
        [ sh:class :Fish ]
        [ sh:class :Meat ]
        [ sh:class :Vegetables ]
        [ sh:class :Pasta ]
        [ sh:class :Other ]
    ) ;
    sh:message "Ingredients must belong to one of
    the defined subclasses (Dairy, Egg, Fish, Meat,
    Vegetables, Pasta, or Other)" ;
    sh:severity sh:Warning .
```

This shape is designed to validate the classification of instances of the class *:Ingredient.* It uses the *sh:or* constraint to ensure that each ingredient belongs to at least one of a predefined set of subclasses: *:Dairy, :Egg, :Fish, :Meat, :Vegetables, :Pasta, or :Other.* If an ingredient does not fall into any of these categories, a warning message is generated: *"Ingredients must belong to one of the defined subclasses (Dairy, Egg, Fish, Meat, Vegetables, Pasta, or Other)"*. The severity level is set to *sh:Warning*, indicating that this constraint is advisory rather than strictly enforced. This shape helps maintain a consistent classification structure within the ontology.

4.4 SWRL Rules

SWRL (Semantic Web Rule Language) [7] is a powerful extension of OWL (Web Ontology Language) that allows for the expression of complex rules and logical inferences within an ontology. These rules follow an "if-then" structure and enable reasoning about relationships and properties that go beyond what is possible with OWL axioms alone. In this project, SWRL rules are used to enhance semantic capabilities by automatically inferring new knowledge about dishes and ingredients, such as dietary classifications or validation constraints. This section presents the set of SWRL rules designed to support intelligent reasoning within the domain of recipe management.
These are the rules that we implemented:

- Infer which ingredients are gluten-free
  `:Ingredient(?x) ∧ :isGlutenFree(?x, true) → :glutenFreeIngredient(?x)`
- Infer which ingredients contain gluten
  `:Ingredient(?x) ∧ :isGlutenFree(?x, false) → :glutenIngredient(?x)`
- Infer which ingredients are lactose free
  `:Ingredient(?x) ∧ :isLactoseFree(?x, true) → :lactoseFreeIngredient(?x)`
- Infer which ingredients contains lactose
  `:Ingredient(?x) ∧ :isLactoseFree(?x, false) → :lactoseIngredient(?x)`
- Infer which ingredients are vegetarian
  `:Ingredient(?x) ∧ :isVegetarian(?x, true) → :vegetarianIngredient(?x)`
- Infer which ingredients are carnivore
  `:Ingredient(?x) ∧ :isVegetarian(?x, false) → :canivorIngredient(?x)`
- Infer which dishes contains gluten
  `:glutenIngredient(?x) ∧ :hasIngredient(?y, ?x) → :glutenMeal(?y)`
- Infer which dishes contain lactose
  `:lactoseIngredient(?x) ∧ :hasIngredient(?y, ?x) → :lactoseMeal(?y)`
- Infer which dishes are carnivore
  `:carnivorIngredient(?x) ∧ :hasIngredient(?y, ?x) → :canivorMeal(?y)`
- Infer a reverse rule for *hasIngredient*
  `:hasIngredient(?x, ?y) → :isInMeal(?y, ?x)`

- Infer which are the ingredients with a low amount of calories
  `:Ingredient(?x) ∧ :hasCalories(?x, ?cal) ∧ swrlb:lessThan(?cal, 101) → :lowCaloriesIngredient(?x)`
- Infer which are the ingredients with a medium amount of calories
  `:Ingredient(?x) ∧ :hasCalories(?x, ?cal) ∧ swrlb:greaterThan(?cal, 100) ∧ swrlb:lessThan(?cal, 200) → :mediumCaloriesIngredient(?x)`
- Infer which are the ingredients with a high amount of calories
  `:Ingredient(?x) ∧ :hasCalories(?x, ?cal) ∧ swrlb:greaterThan(?cal, 199) → :highCaloriesIngredient(?x)`

5. Agile and Ontology-based Meta-Modelling

This chapter explores the agile and ontology-based meta-modeling approach, a modern methodology aimed at improving the flexibility and domain-relevance of modeling languages. By combining agile development practices with ontology-driven structures, this method provides a dynamic and adaptable framework for the creation and evolution of complex models

5.1 AOAME

To complete the second task of the project—focused on agile and ontology-based meta-modeling we employed AOAME [1]., a tool specifically developed for creating and managing Enterprise Knowledge Graph (EKG) schemas. AOAME enables the structured organization of domain-specific knowledge, supporting effective analysis, reasoning, and integration across diverse data sources. It offers flexible meta-modeling capabilities through operators that dynamically modify constructs and automatically generate SPARQL queries to update the triplestore, ensuring alignment between the model and underlying data. Built using Jena Fuseki and Java libraries, AOAME is highly adaptable to real-world scenarios. In our case, it was instrumental in customizing BPMN 2.0, allowing us to extend existing classes and properties to meet the specific needs of our project.

BPMN

BPMN (Business Process Model and Notation) [3] is a standardized graphical notation that allows for the modeling of business processes in a clear and understandable way. Widely adopted across industries, BPMN enables stakeholders to visualize, analyze, and improve organizational workflows. It plays a key role in bridging the gap between technical developers and business analysts.

5.1.1 Our BPMN model

This BPMN 2.0 diagram illustrates a menu customization process designed to deliver personalized food recommendations based on user preferences.
The process begins on the client side, where the user initiates the interaction by scanning a QR code, which triggers the workflow. The user is then prompted to enter their food preferences, such as dietary restrictions, favorite ingredients, or allergies. Once entered, these preferences are sent to the server, where the server-side process takes over. The server receives the preferences and uses them to generate a customized menu, potentially leveraging a knowledge base or ontology to ensure accurate and relevant results.

After the menu is created, it is sent back to the client, who can then view the personalized recommendations. This interaction model ensures a dynamic and user-centric approach to menu selection, allowing real-time customization based on individual needs and preferences, and it may serve as part of a larger semantic web or intelligent food service system.

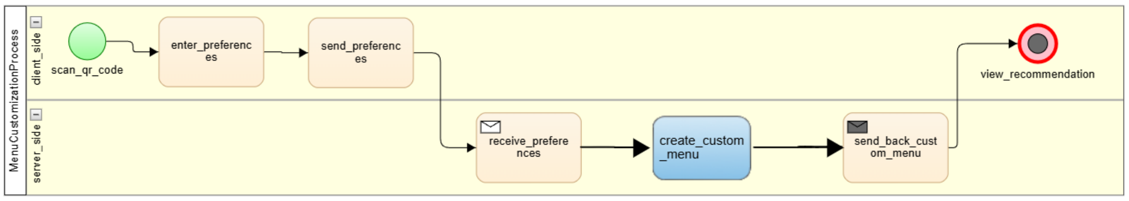
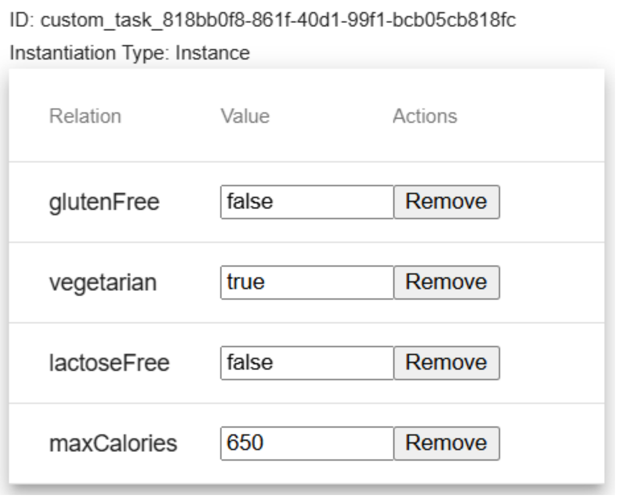


Figura 5.1: BPMN model that describes the process of personalization of a menu in a restaurant.

The extended class *create_custom_menu* is an extended class of Task and includes the data properties shown in Figure 5.2.
The properties **vegetarian**, **glutenFree**, and **lactoseFree** are all boolean type, while **maxCalories** is an integer.


ID: custom_task_818bb0f8-861f-40d1-99f1-bcb05cb818fc
Instantiation Type: Instance

Relation	Value	Actions
glutenFree	false	Remove
vegetarian	true	Remove
lactoseFree	false	Remove
maxCalories	650	Remove

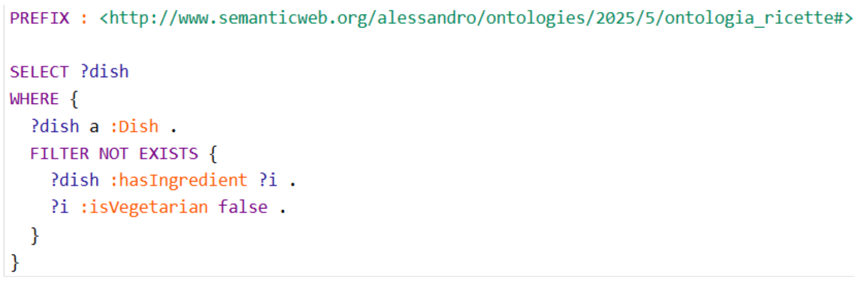
Figura 5.2: Example of the properties used. The client is a vegetarian who wants to eat no more than 650 kcal per dish.

5.2 Jena Fuseki

Jena Fuseki [2] is a SPARQL server that forms part of the Apache Jena framework. It is designed to host RDF (Resource Description Framework) data and provides a robust platform for querying and managing this data using SPARQL. Fuseki supports both SPARQL queries (to read data) and SPARQL updates (to modify data). It can operate as a standalone server with HTTP endpoints for accessing and updating RDF data, or be embedded within applications. This versatility makes Fuseki a valuable tool for building semantic web solutions and managing knowledge graph systems.

5.2.1 Queries in Jena Fuseki

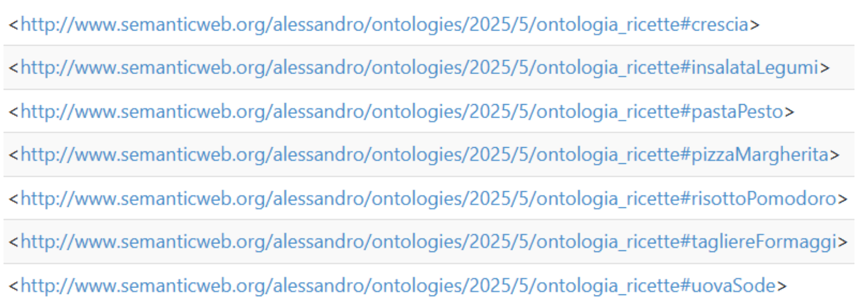
On the Jena Fuseki server, we defined some queries to perform some actions in order to check that everything was working accordingly to what was required.
The purpose of the **first query** (see figure 5.3 we executed was to check if our ontology had been correctly uploaded to the server. We did it by querying our ontology to return all the vegetarian dishes.



```
PREFIX : <http://www.semanticweb.org/alessandro/ontologies/2025/5/ontologia_ricette#>

SELECT ?dish
WHERE {
  ?dish a :Dish .
  FILTER NOT EXISTS {
    ?dish :hasIngredient ?i .
    ?i :isVegetarian false .
  }
}
```

Figura 5.3: Query to get all vegetarian dishes from the ontology

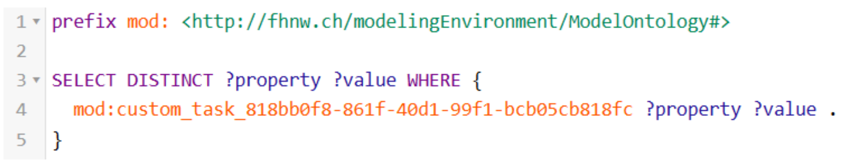
The result (see figure 5.4), as expected, meant that the ontology was uploaded and queried correctly.



```
<http://www.semanticweb.org/alessandro/ontologies/2025/5/ontologia_ricette#crescia>
<http://www.semanticweb.org/alessandro/ontologies/2025/5/ontologia_ricette#insalataLegumi>
<http://www.semanticweb.org/alessandro/ontologies/2025/5/ontologia_ricette#pastaPesto>
<http://www.semanticweb.org/alessandro/ontologies/2025/5/ontologia_ricette#pizzaMargherita>
<http://www.semanticweb.org/alessandro/ontologies/2025/5/ontologia_ricette#risottoPomodoro>
<http://www.semanticweb.org/alessandro/ontologies/2025/5/ontologia_ricette#tagliereFormaggi>
<http://www.semanticweb.org/alessandro/ontologies/2025/5/ontologia_ricette#uovaSode>
```

Figura 5.4: Output of the first query

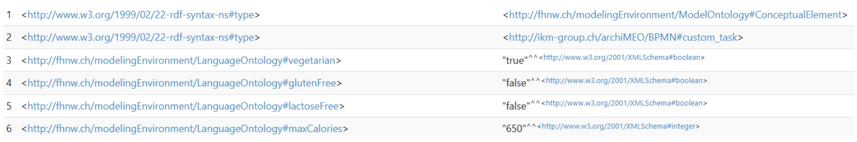
The **second query** (see figure 5.5) was used to check the correct interaction between AOAME and Jena Fuseki, by retrieving the properties from the *create_custom_menu* object we talked about in chapter 5.1.1.
If needed, the ID is `custom_task_818bb0f8-861f-40d1-99f1-bcb05cb818fc`. The



```
prefix mod: <http://fhnw.ch/modelingEnvironment/ModelOntology#>

SELECT DISTINCT ?property ?value WHERE {
  mod:custom_task_818bb0f8-861f-40d1-99f1-bcb05cb818fc ?property ?value .
}
```

Figura 5.5: Query to get the properties from the custom object in AOAME

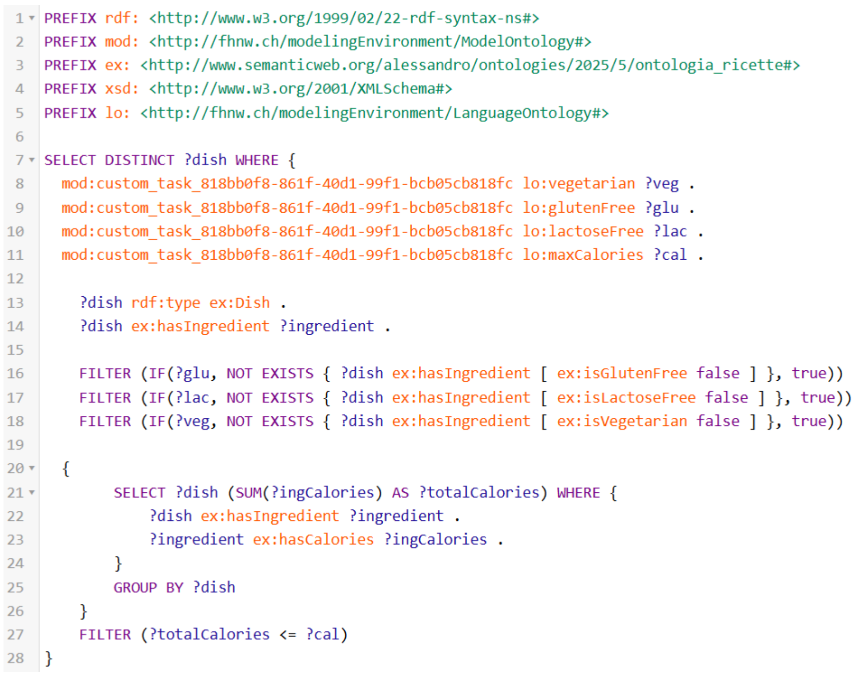
result (see figure 5.6) includes, as expected, the various properties: *vegetarian*, *glutenFree*, *lactoseFree*, and *maxCalories*. Additionally, two rows with the property *rdf:type* appear in the output. These indicate that the resource is classified both as a *ConceptualElement* from the *ModelOntology* and as a *custom_task* from the *archiMEO/BPMN* ontology.
The presence of `rdf:type` is expected because, in RDF, it is standard practice for a resource to belong to one or more classes. Since our query does not filter out any specific



```
1  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>                         <http://fhnw.ch/modelingEnvironment/ModelOntology#ConceptualElement>
2  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>                         <http://ikm-group.ch/archiMEO/BPMN#custom_task>
3  <http://fhnw.ch/modelingEnvironment/LanguageOntology#vegetarian>          "true"^^<http://www.w3.org/2001/XMLSchema#boolean>
4  <http://fhnw.ch/modelingEnvironment/LanguageOntology#glutenFree>          "false"^^<http://www.w3.org/2001/XMLSchema#boolean>
5  <http://fhnw.ch/modelingEnvironment/LanguageOntology#lactoseFree>         "false"^^<http://www.w3.org/2001/XMLSchema#boolean>
6  <http://fhnw.ch/modelingEnvironment/LanguageOntology#maxCalories>         "650"^^<http://www.w3.org/2001/XMLSchema#integer>
```

Figura 5.6: Output of the second query

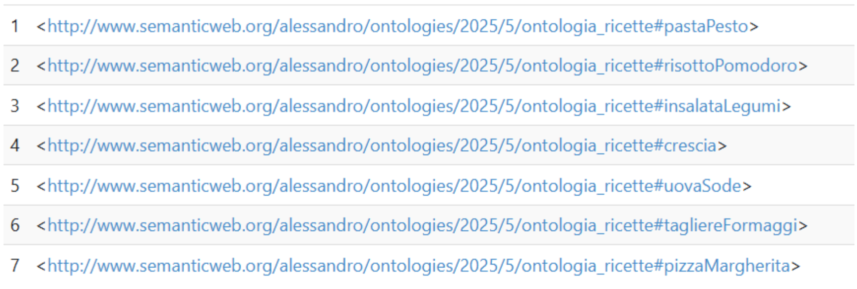
property, it returns all available information linked to the resource, including its type declarations.
The **final query** (see figure 5.7) was used to obtain the menu tailored to the needs of the client. It takes the values of the four properties of the custom object seen in figure 5.2 and assigns them to as many variables that then uses to filter out any non-compliant ingredient. It then sums the calories of the single ingredients grouped by dishes to obtain



```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mod: <http://fhnw.ch/modelingEnvironment/ModelOntology#>
PREFIX ex: <http://www.semanticweb.org/alessandro/ontologies/2025/5/ontologia_ricette#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX lo: <http://fhnw.ch/modelingEnvironment/LanguageOntology#>

SELECT DISTINCT ?dish WHERE {
  mod:custom_task_818bb0f8-861f-40d1-99f1-bcb05cb818fc lo:vegetarian ?veg .
  mod:custom_task_818bb0f8-861f-40d1-99f1-bcb05cb818fc lo:glutenFree ?glu .
  mod:custom_task_818bb0f8-861f-40d1-99f1-bcb05cb818fc lo:lactoseFree ?lac .
  mod:custom_task_818bb0f8-861f-40d1-99f1-bcb05cb818fc lo:maxCalories ?cal .

    ?dish rdf:type ex:Dish .
    ?dish ex:hasIngredient ?ingredient .

    FILTER (IF(?glu, NOT EXISTS { ?dish ex:hasIngredient [ ex:isGlutenFree false ] }, true))
    FILTER (IF(?lac, NOT EXISTS { ?dish ex:hasIngredient [ ex:isLactoseFree false ] }, true))
    FILTER (IF(?veg, NOT EXISTS { ?dish ex:hasIngredient [ ex:isVegetarian false ] }, true))

  {
        SELECT ?dish (SUM(?ingCalories) AS ?totalCalories) WHERE {
            ?dish ex:hasIngredient ?ingredient .
            ?ingredient ex:hasCalories ?ingCalories .
        }
        GROUP BY ?dish
    }
    FILTER (?totalCalories <= ?cal)
}
```

Figura 5.7: Query to obtain the final personalized menu

the total amount of calories for all the dishes. In the end, it filters one last time to rule out the dishes with an amount of calories exceeding the maximum allowed value.
For this particular query, we used the setting seen in figure 5.2, and obtained the results of figure 5.8, that, as expected, satisfy all the enforced conditions.



```
1  <http://www.semanticweb.org/alessandro/ontologies/2025/5/ontologia_ricette#pastaPesto>
2  <http://www.semanticweb.org/alessandro/ontologies/2025/5/ontologia_ricette#risottoPomodoro>
3  <http://www.semanticweb.org/alessandro/ontologies/2025/5/ontologia_ricette#insalataLegumi>
4  <http://www.semanticweb.org/alessandro/ontologies/2025/5/ontologia_ricette#crescia>
5  <http://www.semanticweb.org/alessandro/ontologies/2025/5/ontologia_ricette#uovaSode>
6  <http://www.semanticweb.org/alessandro/ontologies/2025/5/ontologia_ricette#tagliereFormaggi>
7  <http://www.semanticweb.org/alessandro/ontologies/2025/5/ontologia_ricette#pizzaMargherita>
```

Figura 5.8: Output of the final query

6. Conclusions

6.1 Christian Bonsignore

Tackling the same assignment from the different perspectives given by the multiple tasks was a great way to work on my flexibility and adapt to diverse problem-solving paradigms. While the project was challenging (sometimes more due to the software required than the core task itself) it also provided valuable insights and growth opportunities.

- **Trisotech DMN** is a powerful instrument to let the user express a decision logic in a visual and structured way. It was fun to find such a way to represent knowledge and experiment with it, but the experience was dampened by the lack of useful documentation found online, the intricateness and complexity of use in some aspects of the platform, and the untapped potential of having to avoid more useful concepts like complex list handling and iteration.

- **PROLOG** was by far the most enjoyable and rewarding part. Already having experience in programming made it an easier instrument to understand, and the abundance of documentation, explanations, and tutorials was the icing on the cake. In my opinion, sacrificing the more visual approach was what set it apart, making it easier to enforce and infer rules, to use iteration, and to achieve the wanted outcome.

- **Protege** was in my opinion the best compromise between visual and declarative representation, with a variety of interesting functionalities related to it. Creating classes, subclasses, and instances contributed to create a very structured readability with the possibility of a real graphical visualization (OntoGraf). It required a bit more effort than PROLOG at first, for example in the SPARQL queries, but after that it was quite straightforward and engaging.

- **AOAME** was honestly frustrating. The cascade of errors and malfunctions, combined with bad documentation and an overly convoluted process of use, rendered it unusable. On the other hand, once the graph was created, **Jena Fuseki** confirmed my opinions on SPARQL: a powerful, useful and rewarding tool to achieve your goal.

6.2 Alessandro Quercetti

This project has explored a set of paradigms and technologies for modeling, reasoning, and validating knowledge within the domain of dietary preferences and meal planning. Through the combination of all the technologies, we have approached the same problem from different perspectives:

- The use of decision tables allowed us to express complex decision logic in a structured format. This approach was especially effective in representing user preferences, such as diets restrictions or excluded ingredients for determining appropriate meal suggestions. The use of Trisotech provided a visual interface for modeling decision flows. However, we encountered certain limitations, such as the inability to dynamically remove elements from lists. Despite these constraints, this method was useful for communicating decision logic clearly to both technical and non-technical users.
- The use of Prolog offered a declarative and expressive way to encode and reason over facts, rules, and relationships among meals and ingredients. Thanks to Prolog's recursive reasoning, we were able to define dietary filters, compute calorie totals, and verify user constraints efficiently. Personally, this was my favorite part of the project and I found this approach to be highly rewarding, offering both computational power and human-readable logic.
- By developing an OWL ontology using Protégé, we formalized the domain knowledge in graph format. The ontology captured classes such as Dish and Ingredient, along with some linked properties. We tried to create a non-redundant structure, perhaps this decision led us to create an ontology that was not too elaborate, but still appropriate to represent all the information we thought was necessary for this situation.
- Unfortunately, Aoame was the sore point of the project, both in terms of installation and actual use. Initially, the installation guide seemed very intuitive to me, but I immediately found an error that we later understood could have been ignored. Furthermore, the creation of a more complex model was very complicated, since bugs often occurred that slowed down or prevented the normal execution of the program.

Overall, I am happy with how the course topics were addressed, especially the organization and how they were explained. Some implementation steps were difficult for us, but I still felt positively stimulated.

Bibliografia

[1] And. ≪An Agile and Ontology-based Meta-Modelling Approach for the Design and Maintenance of Enterprise Knowledge Graph Schemas≫. In: *EMISA Journal* 19 (2019). DOI: 10.18417/emisa.19.6. URL: https://emisa-journal.org/emisa/article/view/310.

[2] *Apache Jena Fuseki Documentation.* https://jena.apache.org/documentation/fuseki2/. https://jena.apache.org/documentation/fuseki2/.

[3] *Business Process Model and Notation (BPMN).* https://www.bpmn.org/. https://www.bpmn.org/.

[4] M. A. Musen. ≪The Protégé Project: A Look Back and a Look Forward≫. In: *AI Matters* 1.4 (2015). DOI: 10.1145/2557001.25757003.

[5] *Shapes Constraint Language (SHACL).* https://www.w3.org/TR/shacl/. https://www.w3.org/TR/shacl/. 2017.

[6] *SPARQL 1.1 Overview.* https://www.w3.org/TR/sparql11-overview/. https://www.w3.org/TR/sparql11-overview/. 2013.

[7] *SWRL: A Semantic Web Rule Language Combining OWL and RuleML.* https://www.w3.org/Submission/SWRL/. https://www.w3.org/Submission/SWRL/.