



University of Camerino

---

SCHOOL OF SCIENCE AND TECHNOLOGY

Master Degree in Computer Science (Class LM-18)

Knowledge Engineering and Business Intelligence

# Intelligent Meal Suggestion System for Restaurants

## STUDENTS

**Sofia Scattolini**

**Elia Toma**

## SUPERVISORS

**Prof. Knut Hinkelmann**

**Prof. Emanuele Laurenzi**

---

A.Y. 2024/2025



# Abstract

This document presents a knowledge-based system designed to provide personalized menu recommendations in a restaurant setting. The system adapts dynamically to different guest profiles, such as vegetarians, individuals with allergies (such as lactose or gluten intolerance), and calorie-conscious customers.

Three knowledge representation approaches are explored: Decision Tables with Decision Requirements Diagrams (DRDs), Prolog-based logic programming, and an ontology/knowledge graph approach utilizing SWRL rules, SPARQL queries, and SHACL constraints.

Additionally, an ontology-driven extension to BPMN 2.0 is proposed to support meal recommendation workflows by integrating semantic reasoning into business process modeling.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Project Description . . . . .	1
1.2	Project Tasks . . . . .	1
<b>2</b>	<b>Knowledge-Based Solutions</b>	<b>3</b>
2.1	Decision Model . . . . .	3
2.1.1	Decision Tables . . . . .	3
2.1.2	Literal Expression . . . . .	6
2.1.3	Trisotech Simulation . . . . .	7
2.2	Prolog-based Solution . . . . .	8
2.2.1	Prolog Testing . . . . .	11
2.3	Ontology Engineering . . . . .	13
2.3.1	Ontology Modeling . . . . .	13
2.3.2	SWRL Rules . . . . .	14
2.3.3	SPARQL Queries . . . . .	17
2.3.4	SHACL Shape . . . . .	21
2.3.5	Knowledge Graphs . . . . .	22
<b>3</b>	<b>Agile and Ontology-Based Meta-Modelling</b>	<b>25</b>
3.1	AOAME . . . . .	25
3.2	BPMN 2.0 . . . . .	25
3.2.1	BPMN Process Description . . . . .	26
3.2.2	SPARQL Query Execution . . . . .	28
<b>4</b>	<b>Conclusions</b>	<b>35</b>
4.1	Elia Toma - Conclusions . . . . .	35
4.2	Sofia Scattolini - Conclusions . . . . .	36



# List of Figures

2.1	DMN diagram showing decision model structure . . . . .	4
2.2	Partial view of the decide on ingredients decision table . . . . .	4
2.3	Partial view of the decide on meals decision table . . . . .	5
2.4	Ingredients List Literal Expression . . . . .	6
2.5	DMN First Simulation . . . . .	7
2.6	DMN Second Simulation . . . . .	8
2.7	Prolog Testing Scenario 1: Meals recommended for a vegetarian guest without calorie or allergy constraints . . . . .	11
2.8	Prolog Testing Scenario 2: Meals recommended for a vegetarian guest with a calorie limit . . . . .	12
2.9	Core ontology classes . . . . .	13
2.10	Ontology object properties . . . . .	14
2.11	Ontology data properties . . . . .	15
2.12	Inferred meal–guest incompatibilities . . . . .	16
2.13	Single meal–guest incompatibility . . . . .	16
2.14	SPARQL query and results for vegetarian meals execution on GraphDB and Protégé. . . . .	17
2.15	SPARQL query and results for calorie-conscious meals execution on GraphDB and Protégé. . . . .	18
2.16	SPARQL query and results for lactose free meals execution on GraphDB and Protégé. . . . .	19
2.17	SPARQL query and results for calorie-conscious and vegetarian meals execution on GraphDB and Protégé. . . . .	20
2.18	Result of SHACL validation . . . . .	22
2.19	Knowledge graph for lactose-intolerant guests . . . . .	23
2.20	Knowledge graph for vegetarian guests . . . . .	23
3.1	BPMN model of the personalized menu recommendation process . . . . .	26
3.2	Creation of the <i>Suggest Meals</i> task in the BPMN model . . . . .	27
3.3	Guest preference properties for the <i>Suggest Meals</i> task . . . . .	27
3.4	SPARQL query executed by the ‘Suggest Meals’ task in Jena Fuseki . . . . .	28
3.5	Configuration of the <i>vegetarian</i> attribute in the <i>Suggest Meals</i> extended task . . . . .	30
3.6	SPARQL query result showing meals compatible with a vegetarian pref- erence . . . . .	30

3.7	Configuration of the <i>gluten_intolerant</i> attribute in the <i>Suggest Meals</i> extended task . . . . .	31
3.8	SPARQL query result showing meals compatible with a gluten intolerant preference . . . . .	31
3.9	Configuration of the <i>calorie_conscious</i> attribute in the <i>Suggest Meals</i> extended task . . . . .	32
3.10	SPARQL query result showing meals compatible with a calorie conscious preference . . . . .	32
3.11	Configuration of the <i>vegetarian</i> and <i>calorie_conscious</i> attributes in the <i>Suggest Meals</i> extended task . . . . .	33
3.12	SPARQL query result showing meals compatible with both vegetarian and calorie conscious preferences . . . . .	33



# Listings

2.1	Prolog base facts . . . . .	9
2.2	Prolog predicates defining ingredients excluded by dietary restrictions .	9
2.3	Vegetarian meal rule . . . . .	10
2.4	Gluten intolerant meal rule . . . . .	10
2.5	Lactose intolerant meal rule . . . . .	10
2.6	Determine if a meal is calorie-conscious . . . . .	10
2.7	Mapping profiles to meal rules . . . . .	10
2.8	Meal recommendation based on dietary profiles . . . . .	11
2.9	Retrieve Vegetarian Meals . . . . .	17
2.10	Retrieve Calorie-Conscious Meals . . . . .	18
2.11	Retrieve Lactose Free Meals . . . . .	19
2.12	Retrieve Calorie-Conscious Meals . . . . .	20
2.13	SHACL Shape for Guest . . . . .	21
2.14	SHACL Shape for Ingredient . . . . .	21
2.15	SHACL Shape for Meal . . . . .	21

# 1. Introduction

This document describes the work carried out for the Knowledge Engineering and Business Intelligence project entitled *Personalized Menu Recommendation System*. The objective of this project is to design and implement a system capable of dynamically generating a personalized digital menu for restaurant customers, based on their dietary preferences and specific nutritional requirements.

All solutions developed for this project and discussed in the following chapters are available at the GitHub repository [1]:  
[https://github.com/Elia-Toma/KEBI-personalized\\_menu](https://github.com/Elia-Toma/KEBI-personalized_menu).

## 1.1 Project Description

Following the pandemic, many restaurants adopted digital menus accessible via QR codes. This solution improves hygiene and reduces physical contact, but it also introduces a new challenge: limited screen space makes it difficult for customers to quickly browse and compare meals. This issue is especially relevant for customers with specific dietary needs or nutritional goals.

Each guest may have different preferences and restrictions. Some may follow a vegetarian diet, others may be lactose or gluten intolerant, while others may want to reduce calorie intake. Ideally, the digital menu should automatically adapt to each customer by displaying only the meals that meet their personal requirements, simplifying the browsing experience and enhancing customer satisfaction.

The main goal of this project is to represent this knowledge in a structured and machine-readable format, and to develop a system capable of filtering out unsuitable meals and recommending appropriate ones. To achieve this, various approaches are used: rule-based decision tables, logic programming with Prolog, and ontology modeling with OWL and SWRL rules. The project also demonstrates how these models can be integrated within graphical business process models using AOAME.

## 1.2 Project Tasks

The project consists of several tasks, each building upon the previous one:

1. **Knowledge Acquisition and Formalization:** Define the core knowledge about meals, ingredients, guest profiles, and restrictions. This includes representing data such as calories, dietary preferences (e.g., vegetarian), and relationships among these elements (e.g., a meal consists of multiple ingredients).
2. **Decision Tables:** Use DMN tables to represent part of the decision logic. This helps to keep the rules clear, easy to understand, and easy to update when needed.

3. **Prolog Implementation:** Translate the structured knowledge into Prolog facts and rules. This allows reasoning tasks like calculating the total calories of a meal or selecting meals based on the guest's profile.
4. **Ontology Design and Reasoning:** Create an ontology to formalize the domain knowledge and define logical rules (SWRL) to check if a meal is suitable for a guest. This makes it possible to answer questions like which meals are appropriate for different guest profiles.
5. **Graphical Modeling with AOAME:** Extend BPMN models with ontology-based elements, such as a "Suggest Personalized Meal" task. This task includes the specific logic and queries related to the domain.

Each of these tasks offers a different perspective and modeling approach for the same problem. The final result is a complete framework for knowledge-based meal recommendations, which can be used in smart restaurant systems and personalized digital interfaces.

## 2. Knowledge-Based Solutions

This chapter describes the different knowledge-based approaches used to support the creation of personalized menus. Each method represents knowledge in a different way and helps to recommend meals based on the preferences and restrictions of the guests.

The main model (made up of meals, ingredients, and guest profiles) is reused across all approaches but represented in different formats: Decision Tables with DRDs (Decision Requirements Diagrams), logic programming in Prolog, ontology modeling, and extended BPMN 2.0 diagrams using AOAME.

All solutions aim to solve the same problem: selecting meals that match the profile of a guest. This includes dietary needs such as being vegetarian, lactose or gluten intolerant, or limiting calorie intake.

### 2.1 Decision Model

The first knowledge-based solution relies on Decision Tables developed according to the Decision Model and Notation (DMN) standard. These diagrams were developed with the Trisotech Decision Modeler platform [2] and organized in a modular and hierarchical structure through a Decision Requirements Diagram (DRD). The DRD visually represents the structure of the decisions and how they are connected and depend on each other, making the logic easier to understand and manage.

As shown in Figure 2.1, the model includes two Decision Tables and one Literal Expression. The logic works in two steps: first, the ingredients are checked according to the restrictions of the guest, then the meals are filtered according to the allowed ingredients.

This DMN model is designed to determine the most suitable meals for a guest by considering their dietary preferences, food intolerances, and calorie limits.

The following section explains each Decision Table in detail and describes how they work together to support the decision-making process.

#### 2.1.1 Decision Tables

Based on Figure 2.1, the DMN model includes two main Decision Tables, each responsible for a specific step in the decision-making process. Below is a detailed description of these tables, their inputs, and outputs:

- **decide on ingredients:** This Decision Table evaluates each ingredient according to the guest's dietary restrictions. It takes three boolean inputs: `isVegetarian`, `isLactoseIntolerant`, and `isGlutenIntolerant`. Each rule checks whether

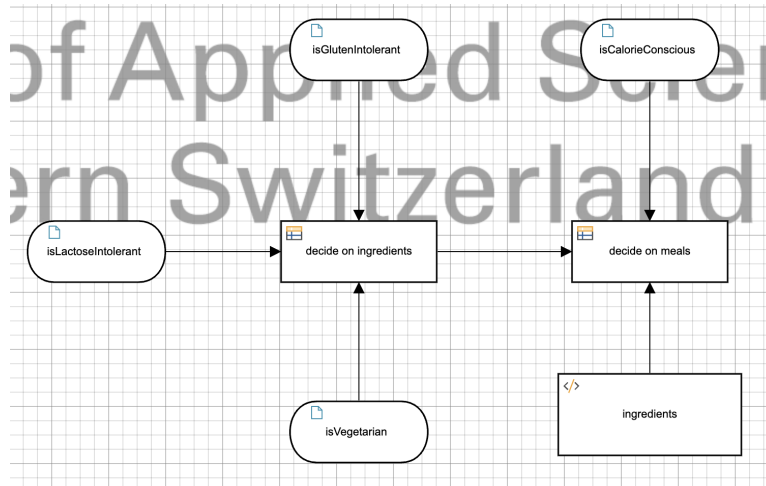


Figure 2.1: DMN diagram showing decision model structure

a particular ingredient is allowed or should be excluded according to these restrictions. The output is a list of ingredients that satisfy the input conditions.

	inputs			outputs	annotations
	isVegetarian	isLactoseIntolerant	isGlutenIntolerant	decide on ingredients	Description
C	Boolean	Boolean	Boolean	Text	
1	false	-	-	"prosciutto_crudo"	
2	false	-	-	"salame"	
3	false	-	-	"mortadella"	
4	false	-	-	"coppa"	
5	-	false	-	"gorgonzola"	
6	-	false	-	"fontina"	
7	-	false	-	"pecorino"	
8	-	-	false	"farina"	

Figure 2.2: Partial view of the decide on ingredients decision table

- **decide on meals:** This Decision Table determines suitable meals by combining the list of filtered ingredients produced by the previous table. It uses the following inputs:
  - The output of the decide on ingredients table (a list of allowed ingredients).
  - A boolean input `isCalorieConscious` indicating whether calorie con-

straints should be applied.

Each rule checks if the required ingredients for a specific meal are present in the list. When `isCalorieConscious` is true, the total calories of the selected ingredients must not exceed a specific limit (such as 600 kcal).

C	decide on ingredients	isCalorieConscious	ingredients	decide on meals	Description
	<i>Text</i>	<i>Boolean</i>	<i>ingredientsType</i>	<i>Text</i>	
1	list contains(?, "mozzarella") and list contains(?, "pomodoro") and list contains(?, "basilico") and list contains(?, "olio_oliva")	true	(?[name="mozzarella"].calories[1] + ?[name="pomodoro"].calories[1] + ?[name="basilico"].calories[1] + ?[name="olio_oliva"].calories[1]) <= 600	"caprese"	
2	list contains(?, "mozzarella") and list contains(?, "pomodoro") and list contains(?, "basilico") and list contains(?, "olio_oliva")	false	-	"caprese"	
3	list contains(?, "prosciutto_crudo") and list contains(?, "salame") and list contains(?, "mortadella") and list contains(?, "coppa")	true	(?[name="prosciutto_crudo"].calories[1] + ?[name="salame"].calories[1] + ?[name="mortadella"].calories[1] + ?[name="coppa"].calories[1]) <= 600	"tagliere salumi"	
4	list contains(?, "prosciutto_crudo") and list contains(?, "salame") and list contains(?, "mortadella") and list contains(?, "coppa")	false	-	"tagliere salumi"	
5	list contains(?, "gorgonzola") and list contains(?, "fontina") and list contains(?, "pecorino")	true	(?[name="gorgonzola"].calories[1] + ?[name="fontina"].calories[1] + ?[name="pecorino"].calories[1]) <= 600	"tagliere formaggi"	
6	list contains(?, "gorgonzola") and list contains(?, "fontina") and list contains(?, "pecorino")	false	-	"tagliere formaggi"	

Figure 2.3: Partial view of the decide on meals decision table

Both decision tables use the **Collect** hit policy. This means that all the rules that match the conditions are collected together, and not only the first one that applies.

- In the `decide on ingredients` table, this policy collects all the ingredients that respect the guest's dietary restrictions. Each rule adds one possible ingredient to the final list of allowed ingredients.
- In the `decide on meals` table, this policy collects all meals that can be prepared using the allowed ingredients from the previous table. If the guest is calorie conscious, the table also checks that the total calories of the meal are below the specified limit (for example, 600 kcal). Only meals that meet both ingredient and calorie conditions are included in the result.

This allows the model to suggest all possible suitable options for the guest.

### 2.1.2 Literal Expression

To streamline the decision-making process, a Literal Expression has been developed to represent the ingredients and their corresponding calorie values. This expression defines a Map structure in which each ingredient name is linked to a numerical value that indicates its calorie content. This approach simplifies the evaluation of calorie restrictions within the decision tables by providing direct access to the caloric information of each ingredient.

Below is an excerpt of the Literal Expression 2.4 that illustrates this mapping.

#### ingredients

*ingredientsType*

```
[
  {"name": "prosciutto_crudo", "calories": 150},
  {"name": "salame", "calories": 200},
  {"name": "mortadella", "calories": 250},
  {"name": "coppa", "calories": 180},
  {"name": "gorgonzola", "calories": 300},
  {"name": "fontina", "calories": 280},
  {"name": "farina", "calories": 340},
  {"name": "pomodoro", "calories": 20},
  {"name": "basilico", "calories": 5},
  {"name": "mozzarella", "calories": 150},
  {"name": "pecorino", "calories": 120},
  {"name": "olio_oliva", "calories": 120},
  {"name": "spaghetti", "calories": 350},
  {"name": "penne", "calories": 360},
  {"name": "riso", "calories": 330},
  {"name": "pollo", "calories": 200},
  {"name": "manzo", "calories": 250},
  {"name": "uova", "calories": 70},
  {"name": "guanciale", "calories": 200},
  {"name": "funghi", "calories": 25},
  {"name": "zucchine", "calories": 15},
  {"name": "vongole", "calories": 180},
  {"name": "gamberetti", "calories": 150},
  {"name": "lattuga", "calories": 15},
  {"name": "cetriolo", "calories": 16},
  {"name": "olive", "calories": 115},
  {"name": "carota", "calories": 41}
]
```

Figure 2.4: Ingredients List Literal Expression

### 2.1.3 Trisotech Simulation

Finally, an example of meal suggestions generated by the model using Trisotech is presented. In this simulation, the following parameters have been set: `isVegetarian` to `true`, while `isLactoseIntolerant` and `isGlutenIntolerant` are set to `false`. The `isCalorieConscious` parameter is also set to `false`.

Trace In Data Out Breakpoints

isVegetarian: true

isLactoseIntolerant: false

isGlutenIntolerant: false

isCalorieConscious: false

Submit Clear

Trace In Data Out Breakpoints

decide on meals

```
[
  "caprese",
  "tagliere formaggi",
  "pizza margherita",
  "pasta vegetariana",
  "insalata mista"
]
```

Save Download

	isVegetarian	isLactoseIntolerant	isGlutenIntolerant	Ingredients	Description
1				"prosciutto_crudo"	
2	false	-	-	"salame"	
3	false	-	-	"mortadella"	
4	false	-	-	"coppa"	
5	-	false	-	"gorgonzola"	
6	-	false	-	"fontina"	
7	-	false	-	"pecorino"	
8	-	-	false	"farina"	
9	-	-	-	"pomodoro"	
10	-	-	-	"basilico"	
11	-	false	-	"mozzarella"	
12	-	-	-	"olio_oliva"	
13	-	-	false	"spaghetti"	

	decide on ingredients	isCalorieConscious	Ingredients	Is	Description
1	list contains? "mozzarella" and list contains? "pomodoro" and list contains? "basilico" and list contains? "olio_oliva"	true	([name="mozzarella"] calories() * 7 + [name="pomodoro"] calories() * 7 + [name="basilico"] calories() * 7 + [name="olio_oliva"] calories()) <= 600	"caprese"	
2	list contains? "mozzarella" and list contains? "pomodoro" and list contains? "basilico" and list contains? "olio_oliva"	false	-	"caprese"	
3	list contains? "prosciutto_crudo" and list contains? "salame" and list contains? "mortadella" and list contains? "coppa"	true	([name="prosciutto_crudo"] calories() * 7 + [name="salame"] calories() * 7 + [name="mortadella"] calories() * 7 + [name="coppa"] calories()) <= 600	"tagliere salumi"	
4	list contains? "prosciutto_crudo" and list contains? "salame" and list contains? "mortadella" and list contains? "coppa"	false	-	"tagliere salumi"	
5	list contains? "gorgonzola" and list contains? "fontina" and list contains? "pecorino"	true	([name="gorgonzola"] calories() * 7 + [name="fontina"] calories() * 7 + [name="pecorino"] calories()) <= 600	"tagliere formaggi"	
6	list contains? "gorgonzola" and list contains? "fontina" and list contains? "pecorino"	false	-	"tagliere formaggi"	
7	list contains? "farina" and list contains? "pomodoro" and list contains? "mozzarella" and list contains? "basilico" and list contains? "olio_oliva"	true	([name="farina"] calories() * 7 + [name="pomodoro"] calories() * 7 + [name="mozzarella"] calories() * 7 + [name="basilico"] calories() * 7 + [name="olio_oliva"] calories()) <= 600	"pizza margherita"	
8	list contains? "farina" and list contains? "pomodoro" and list contains? "mozzarella" and list contains? "basilico" and list contains? "olio_oliva"	false	-	"pizza margherita"	

Figure 2.5: DMN First Simulation

If the `isCalorieConscious` parameter is now set to `true`, meals "Tagliere formaggi" and "Pizza margherita" will no longer be included in the suggested options, as their total calorie content exceeds the specified limit. This result is illustrated in Figure 2.6.



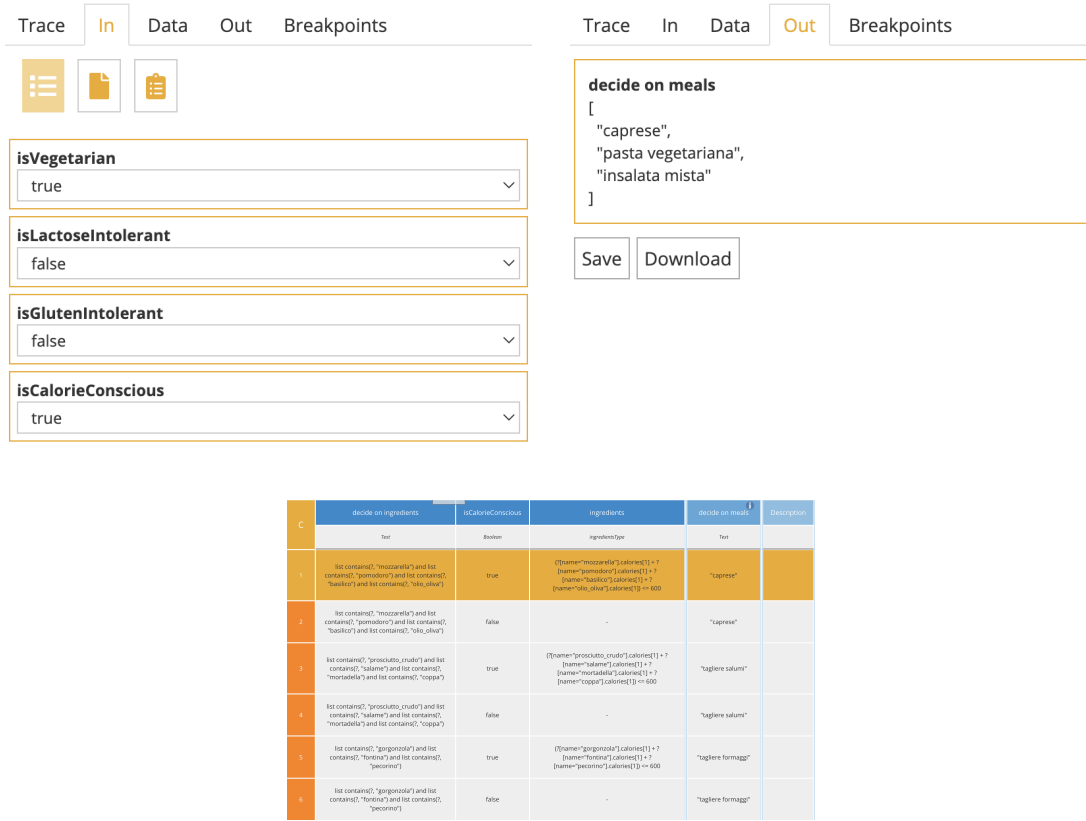


Figure 2.6: DMN Second Simulation

## 2.2 Prolog-based Solution

The second knowledge-based solution uses **Prolog**, a declarative logic programming language that is well suited to represent rule-based knowledge and perform recursive reasoning. Implementation was carried out using the online tool **SWISH-PROLOG** [3].

Compared to the decision table approach, the Prolog solution provides greater flexibility. It allows a direct definition of facts and rules to represent meals, ingredients, dietary restrictions, and calories limits. Thanks to Prolog's pattern matching and rule-based reasoning, it becomes easy to describe how meals are prepared and to check whether they meet all guest requirements, such as food intolerances and calorie restrictions.

In this model, each entity is represented as a set of logical facts:

- **Ingredients** are defined using the unary predicate `ingredient/1`, for example: `ingredient(pomodoro)`.
- **Caloric values** are associated with each ingredient through the binary predicate `calories/2`, for example: `calories(mozzarella, 150)`.
- **Meals** are described using the binary predicate `meal/2`, where each meal has a name and a list of ingredients, for example: `meal(caprese, [mozzarella, pomodoro, basilico, olio_oliva])`.

```

1  % Ingredients: ingredient (Name)
2  ingredient(mozzarella) .
3  ingredient(pomodoro) .
4  ...
5
6  % Calories
7  calories(mozzarella, 150) .
8  calories(pomodoro, 20) .
9  ...
10
11 % Meals: meal (Name, [List of Ingredients])
12 meal(caprese, [mozzarella, pomodoro, basilico, olio_oliva]) .
13 meal(pizza_margherita, [farina, mozzarella, pomodoro, basilico,
14                        olio_oliva]) .

```

Listing 2.1: Prolog base facts

Dietary rules are modeled by listing the ingredients that are not allowed under specific dietary conditions. These restrictions are expressed using dedicated Prolog predicates:

- `non_vegetarian/1` lists ingredients that are excluded from vegetarian diets.
- `non_lactose/1` includes ingredients that contain lactose and are not suitable for lactose-intolerant guests.
- `non_gluten/1` includes ingredients that contain gluten and are not suitable for gluten-intolerant guests.

These predicates make it easy to filter ingredients and meals according to the guest's dietary profile.

```

1  % Ingredients not compatible with a vegetarian diet
2  non_vegetarian(prosciutto_crudo) .
3  non_vegetarian(pollo) .
4  ...
5
6  % Ingredients not compatible with a lactose intolerance
7  non_lactose(mozzarella) .
8  ...
9
10 % Ingredients not compatible with a gluten intolerance
11 non_gluten(spaghetti) .
12 non_gluten(penne) .
13 non_gluten(farina) .

```

Listing 2.2: Prolog predicates defining ingredients excluded by dietary restrictions

Based on the previously defined predicates, the following rules are used to check whether a meal meets the guest's dietary requirements:

- `is_vegetarian/1`, `is_lactose_free/1`, and `is_gluten_free/1` check that none of the meal's ingredients violate the specific dietary restriction.

- `meal_calories/2` calculates the total calories of a meal by summing the calories of all its ingredients.
- `is_calorie_conscious/1` checks whether the total calorie count of a meal stays below a defined threshold, specified by the predicate `calorie_threshold/1`.

Below are the Prolog rules that implement these checks:

```

1  is_vegetarian(Meal) :-
2      meal(Meal, Ingredients),
3      \+ (member(Ingredient, Ingredients),
4          non_vegetarian(Ingredient)).

```

Listing 2.3: Vegetarian meal rule

```

1  is_gluten_free(Meal) :-
2      meal(Meal, Ingredients),
3      \+ (member(Ingredient, Ingredients),
4          non_gluten(Ingredient)).

```

Listing 2.4: Gluten intolerant meal rule

```

1  is_lactose_free(Meal) :-
2      meal(Meal, Ingredients),
3      \+ (member(Ingredient, Ingredients),
4          non_lactose(Ingredient)).

```

Listing 2.5: Lactose intolerant meal rule

```

1  is_calorie_conscious(Meal) :-
2      meal_calories(Meal, TotalCalories),
3      calorie_threshold(Threshold),
4      TotalCalories <= Threshold.

```

Listing 2.6: Determine if a meal is calorie-conscious

The predicate `fits/2` maps each dietary profile (such as `vegetarian` or `gluten_intolerant`) to the corresponding rule that determines whether a meal satisfies that profile.

The predicate `recommend_meal/2` provides personalized meal recommendations based on one or more dietary profiles. For example, the query `recommend_meal([vegetarian, calorie_conscious], meal)` returns all meals that are both vegetarian and below the calorie limit.

To evaluate multiple dietary constraints, a helper predicate called `all_fit/2` is used. This predicate recursively verifies that a meal satisfies all profiles in a given list.

```

1  fits(vegetarian, Meal) :- is_vegetarian(Meal).
2  fits(carnivore, Meal) :- is_carnivore(Meal).
3  fits(lactose_intolerant, Meal) :- is_lactose_free(Meal).
4  fits(gluten_intolerant, Meal) :- is_gluten_free(Meal).
5  fits(calorie_conscious, Meal) :- is_calorie_conscious(Meal).

```

Listing 2.7: Mapping profiles to meal rules

```

1  % Collect meals that are compliant with all given preferences
2  recommend_meal(Profile, Meal) :-
3      \+ is_list(Profile),
4      fits(Profile, Meal).
5
6  recommend_meal(Profiles, Meal) :-
7      is_list(Profiles),
8      all_fit(Profiles, Meal).
9
10 % Helper to check all profiles fit
11 all_fit([], _).
12 all_fit([P|Ps], Meal) :-
13     fits(P, Meal),
14     all_fit(Ps, Meal).

```

Listing 2.8: Meal recommendation based on dietary profiles

Compared to the DMN-based approach, this Prolog-based solution offers enhanced expressiveness and flexibility. It supports multi-criteria queries, enabling the evaluation of several dietary constraints simultaneously. Moreover, due to Prolog’s native support for recursion and list processing, filtering logic can incorporate complex reasoning patterns that go beyond static decision tables. This makes it possible to implement more dynamic and adaptable recommendation strategies.

### 2.2.1 Prolog Testing

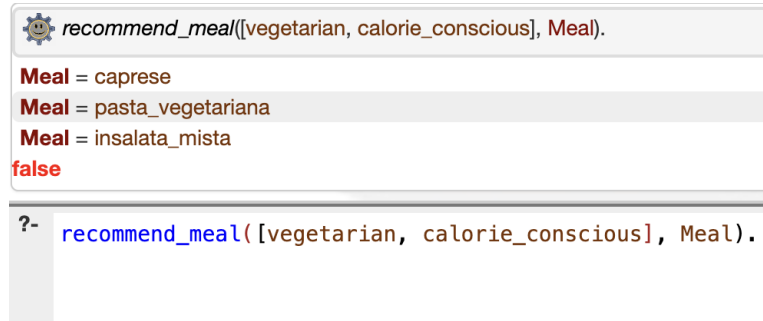
This section presents two test scenarios designed to validate the accuracy of the Prolog-based recommendation system with respect to guest dietary preferences.

In the **first test case**, the guest follows a vegetarian diet but does not have allergies or calorie restrictions. The objective is to verify that the system correctly identifies meals containing no non-vegetarian ingredients, while ignoring other criteria.



Figure 2.7: Prolog Testing Scenario 1: Meals recommended for a vegetarian guest without calorie or allergy constraints

In the **second test case**, the guest is both vegetarian and calorie-conscious. This scenario evaluates the system's ability to apply multiple constraints simultaneously, filtering meals that meet both the vegetarian requirement and the defined calorie threshold.



```
⚙️ recommend_meal([vegetarian, calorie_conscious], Meal).  
  
Meal = caprese  
Meal = pasta_vegetariana  
Meal = insalata_mista  
false  
  
?- recommend_meal([vegetarian, calorie_conscious], Meal).
```

Figure 2.8: Prolog Testing Scenario 2: Meals recommended for a vegetarian guest with a calorie limit

These test cases demonstrate that the Prolog predicates implemented are capable of handling composite diet profiles and generating accurate meal recommendations accordingly.

## 2.3 Ontology Engineering

The third knowledge-based solution is built on an ontology initially developed using Protégé [4], exported in Turtle (.ttl) format, and imported into GraphDB [5] to generate a semantic graph. This ontology-driven approach enables formal reasoning through OWL semantics and supports SPARQL querying to explore structured information about meals, ingredients, and guest dietary preferences.

### 2.3.1 Ontology Modeling

#### Classes

The conceptual model defines the core domain classes: `Guest`, `Meal`, `Ingredient`, and `GuestProfile`. Dietary preferences and calorie-related restrictions are modeled as specialized subclasses, such as `GuestFoodTypePreference` and `GuestCaloriesPreference`.

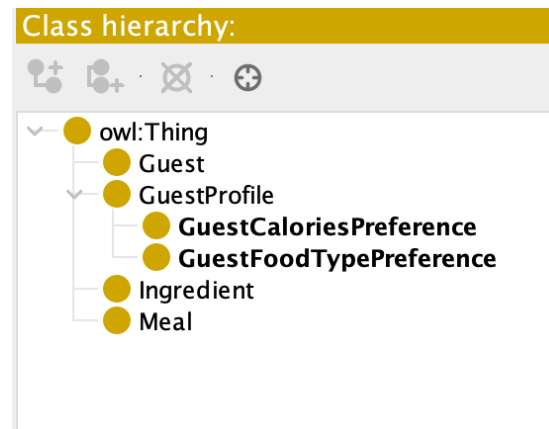


Figure 2.9: Core ontology classes

#### Object Properties

Object properties in the ontology are used to formally define semantic relationships between key domain entities. These properties enable inferencing and support constraint validation across meals, ingredients, and guest profiles. In particular:

- `hasIngredient`: links a `Meal` individual to one or more `Ingredient` individuals, specifying the components of each dish.
- `hasProfile`: associates a `Guest` with a `GuestProfile`, which includes both food-type preferences (such as vegetarian, lactose intolerant) and caloric constraints.
- `ingredientNotCompatibleWithGuestFoodTypePreference`: models a semantic incompatibility between an `Ingredient` and a specific `GuestFoodTypePreference`, such as cheese being incompatible with a lactose-intolerant profile.
- `mealNotCompatibleWithGuestProfile`: indicates that a given `Meal` is not suitable for a specific `GuestProfile`. This property is not asserted directly, but

inferred through SWRL rules that propagate incompatibilities from ingredients to meals, based on guest restrictions.

These object properties are central to the ontology’s reasoning capabilities. They are used in SWRL rules to infer higher-level knowledge (such as filtering out meals incompatible with a guest’s profile) and are also validated through SHACL to ensure logical consistency in the knowledge base.



Figure 2.10: Ontology object properties

## Data Properties

Data properties like `ingredientHasCalories` and `mealHasCalories` are used to store nutritional information about ingredients and meals.

Specifically, in the ontology, data properties link an object (such as an ingredient or a meal) to a literal value, usually a number or text.

- `ingredientHasCalories` associates each ingredient with a numeric value that represents the calories it contains. For example, the ingredient “tomato” might have `ingredientHasCalories` equal to 20, which means it contains 20 calories.
- `mealHasCalories` links a meal (such as “pizza margherita”) to its total calorie count. For example, a pizza margherita might have `mealHasCalories` equal to 800, indicating that the entire meal has 800 calories.

These properties are important because they allow the system to know how many calories are in each ingredient and meal. This helps when filtering or recommending food to guests who care about their calorie intake (for example, guests who want meals under 500 calories).

In summary, these data properties store calorie information in a clear and structured way within the ontology, enabling the system to personalize meal recommendations based on nutritional values.

### 2.3.2 SWRL Rules

Two SWRL rules (Semantic Web Rule Language) are implemented to infer dietary incompatibilities between meals and guest profiles:

- **InferMealIncompatibilityWithGuestProfile:** This rule states that if a meal contains at least one ingredient that conflicts with a guest’s dietary preference

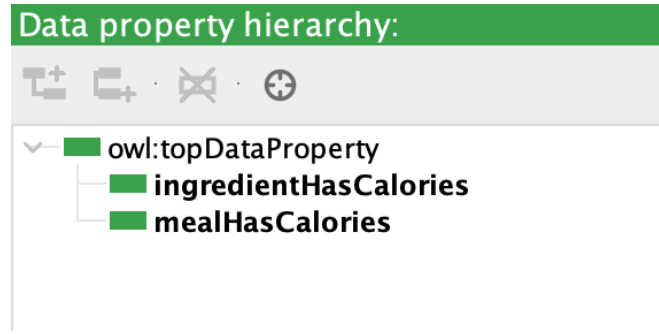


Figure 2.11: Ontology data properties

(such as vegetarian), then the meal is flagged as incompatible with that guest profile. The rule can be expressed as:

$$\begin{aligned} & \text{Meal}(?meal) \wedge \text{hasIngredient}(?meal, ?ingredient) \wedge \\ & \text{ingredientNotCompatibleWithGuestFoodTypePreference}(?ingredient, ?guestPreference) \\ & \rightarrow \text{mealNotCompatibleWithGuestProfile}(?meal, ?guestPreference) \end{aligned}$$

- **CalorieConsciousMealExceedsLimitRule:** This rule infers that if a meal's calorie content exceeds 600 kcal and the guest is classified as `calorie_conscious`, then the meal is considered unsuitable and marked as incompatible. Formally:

$$\begin{aligned} & \text{Meal}(?m) \wedge \text{mealHasCalories}(?m, ?cal) \wedge \\ & \text{swrlb:greaterThan}(?cal, 600) \wedge \text{Guest}(?g) \wedge \\ & \text{hasProfile}(?g, \text{calorie\_conscious}) \rightarrow \\ & \text{mealNotCompatibleWithGuestProfile}(?m, \text{calorie\_conscious}) \end{aligned}$$

These SWRL rules enrich the ontology with inferential knowledge that cannot be represented by simple class hierarchies or property assertions alone. Instead of manually verifying the compatibility of each meal with a guest's dietary preferences, these rules allow a reasoning engine to automatically infer which meals are suitable or not.

This automatic inference classifies meals as compliant or non-compliant based on the presence of incompatible ingredients or the exceeding of the calorie limits, without the need for explicit procedural code.



## SWRL Testing

The SWRL rules were tested using the Pellet reasoner within Protégé. The test involved reasoning with individuals representing meals, ingredients and guests with various dietary preferences.

The reasoner correctly inferred incompatibilities between meals and guest profiles based on ingredient conflicts and calorie limits, as specified by the SWRL rules.

Figure 2.12 shows inferred results that highlight meals flagged as incompatible due to multiple dietary restrictions or calorie excess.

Property assertions: carbonara		
Object property assertions +		
hasIngredient	guanciale	? @ x o
hasIngredient	pecorino	? @ x o
hasIngredient	spaghetti	? @ x o
hasIngredient	uova	? @ x o
mealNotCompatibleWithGuestProfile	calorie_conscious	? @
mealNotCompatibleWithGuestProfile	gluten_intolerant	? @
mealNotCompatibleWithGuestProfile	lactose_intolerant	? @
mealNotCompatibleWithGuestProfile	vegetarian	? @

Figure 2.12: Inferred meal–guest incompatibilities

This second example in Figure 2.13 shows inference with a single incompatibility.

Property assertions: caprese		
Object property assertions +		
hasIngredient	basilico	? @ x o
hasIngredient	mozzarella	? @ x o
hasIngredient	olio_oliva	? @ x o
hasIngredient	pomodoro	? @ x o
mealNotCompatibleWithGuestProfile	lactose_intolerant	? @

Figure 2.13: Single meal–guest incompatibility

### 2.3.3 SPARQL Queries

**SPARQL** queries were used to retrieve and filter data from the ontology according to specific requirements. These queries allow the system to dynamically extract relevant information without procedural programming.

In the following subsection, we will describe some of the queries.

#### SPARQL Queries Examples

**Retrieve Vegetarian Meals** This query returns all meals that are compatible with a vegetarian diet. It checks if the meal does not contain any ingredient that is explicitly marked as incompatible with the vegetarian food preference. If such an ingredient is found, the meal is excluded from the result.

```

1 PREFIX : <http://www.semanticweb.org/user/ontologies/2025/5/
  personalized_menu/>
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3
4 SELECT DISTINCT ?meal WHERE {
5   ?meal rdf:type :Meal .
6
7   FILTER NOT EXISTS {
8     ?meal :hasIngredient ?ingredient .
9     ?ingredient :
      ingredientNotCompatibleWithGuestFoodTypePreference :
      vegetarian .
10  }
11 }
```

Listing 2.9: Retrieve Vegetarian Meals

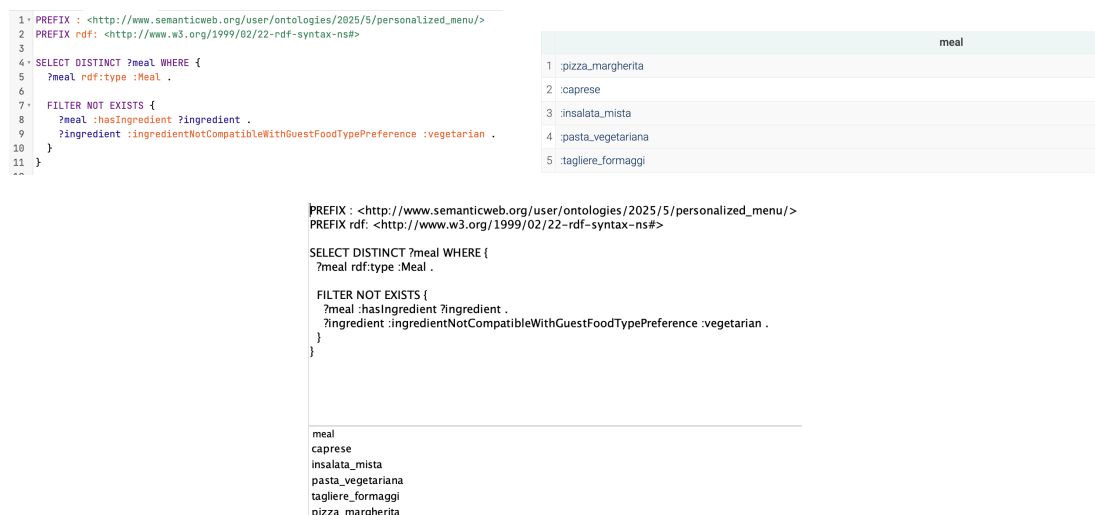


Figure 2.14: SPARQL query and results for vegetarian meals execution on GraphDB and Protégé.

**Retrieve Calorie-Conscious Meals** This query selects all meals whose total calories do not exceed 600 kcal, which is considered the threshold for a calorie-conscious meal. The query sums the calories of each ingredient in a meal and filters out those that exceed the limit.

```

1 PREFIX : <http://www.semanticweb.org/user/ontologies/2025/5/
  personalized_menu/>
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3
4 SELECT ?meal (SUM(?calories) AS ?totalCalories)
5 WHERE {
6   ?meal rdf:type :Meal .
7       :hasIngredient ?ingredient .
8   ?ingredient :ingredientHasCalories ?calories .
9 }
10 GROUP BY ?meal
11 HAVING (SUM(?calories) <= 600)

```

Listing 2.10: Retrieve Calorie-Conscious Meals

```

1 PREFIX : <http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/>
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3
4 SELECT ?meal (SUM(?calories) AS ?totalCalories)
5 WHERE {
6   ?meal rdf:type :Meal ;
7       :hasIngredient ?ingredient .
8   ?ingredient :ingredientHasCalories ?calories .
9 }
10 GROUP BY ?meal
11 HAVING (SUM(?calories) <= 600)
12

```

	meal	\$	totalCalories
1	caprese	"295"	xsd:integer
2	pasta_vegetariana	"540"	xsd:integer
3	insalata_mista	"327"	xsd:integer
4	pollo_grigliato	"320"	xsd:integer
5	risotto_gamberetti	"600"	xsd:integer
6	tagliata_di_manzo	"370"	xsd:integer

```

PREFIX : <http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

```

```

SELECT ?meal (SUM(?calories) AS ?totalCalories)
WHERE {
  ?meal rdf:type :Meal ;
      :hasIngredient ?ingredient .
  ?ingredient :ingredientHasCalories ?calories .
}
GROUP BY ?meal
HAVING (SUM(?calories) <= 600)

```

meal	totalCalories
caprese	"295"^^<http://www.w3.org/2001/XMLSchema#integer>
insalata_mista	"327"^^<http://www.w3.org/2001/XMLSchema#integer>
tagliata_di_manzo	"370"^^<http://www.w3.org/2001/XMLSchema#integer>
pollo_grigliato	"320"^^<http://www.w3.org/2001/XMLSchema#integer>
pasta_vegetariana	"540"^^<http://www.w3.org/2001/XMLSchema#integer>
risotto_gamberetti	"600"^^<http://www.w3.org/2001/XMLSchema#integer>

Figure 2.15: SPARQL query and results for calorie-conscious meals execution on GraphDB and Protégé.

**Retrieve Lactose Free Meals** This query returns all meals that are compatible with guests who are lactose intolerant. It excludes meals containing ingredients explicitly marked as incompatible with the `lactose_intolerant` food preference.

The query structure for retrieving gluten free meals is analogous, differing only in the food preference value, which is `gluten_intolerant` instead of `lactose_intolerant`.

```

1 PREFIX : <http://www.semanticweb.org/user/ontologies/2025/5/
  personalized_menu/>
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3
4 SELECT DISTINCT ?meal WHERE {
5   ?meal rdf:type :Meal .
6
7   FILTER NOT EXISTS {
8     ?meal :hasIngredient ?ingredient .
9     ?ingredient :
10      ingredientNotCompatibleWithGuestFoodTypePreference :
11      lactose_intolerant .
12   }
13 }
```

Listing 2.11: Retrieve Lactose Free Meals

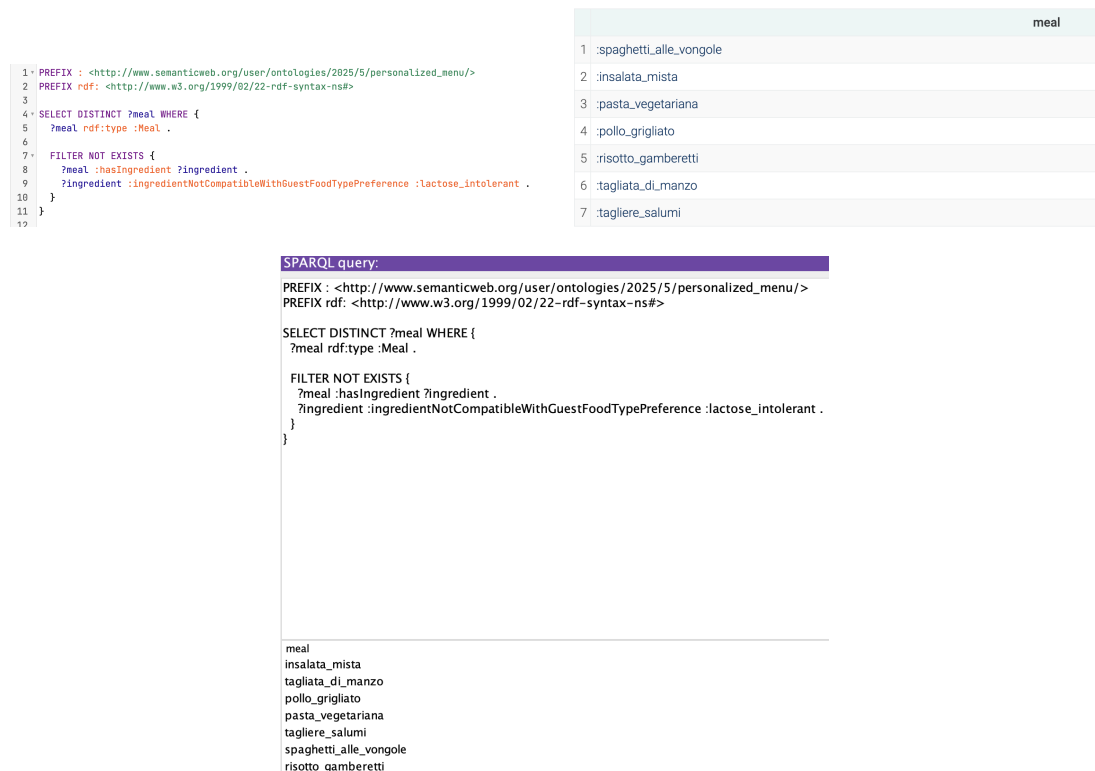


Figure 2.16: SPARQL query and results for lactose free meals execution on GraphDB and Protégé.

**Retrieve Calorie-Conscious and Vegetarian Meals** This query combines two criteria to find vegetarian meals a total calorie count less than or equal to 600 kcal. It excludes meals with ingredients incompatible with the vegetarian preference and filters based on the calorie sum.

The same approach can be extended to combine other preferences, such as gluten-free and lactose-free.

```

1 PREFIX : <http://www.semanticweb.org/user/ontologies/2025/5/
  personalized_menu/>
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3
4 SELECT ?meal (SUM(?calories) AS ?totalCalories)
5 WHERE {
6   ?meal rdf:type :Meal .
7   ?meal :hasIngredient ?ingredient .
8   ?ingredient :ingredientHasCalories ?calories .
9
10  FILTER NOT EXISTS {
11    ?meal :hasIngredient ?ingredient .
12    ?ingredient :
      ingredientNotCompatibleWithGuestFoodTypePreference :
      vegetarian .
13  }
14 }
15 GROUP BY ?meal
16 HAVING (SUM(?calories) <= 600)

```

Listing 2.12: Retrieve Calorie-Conscious Meals

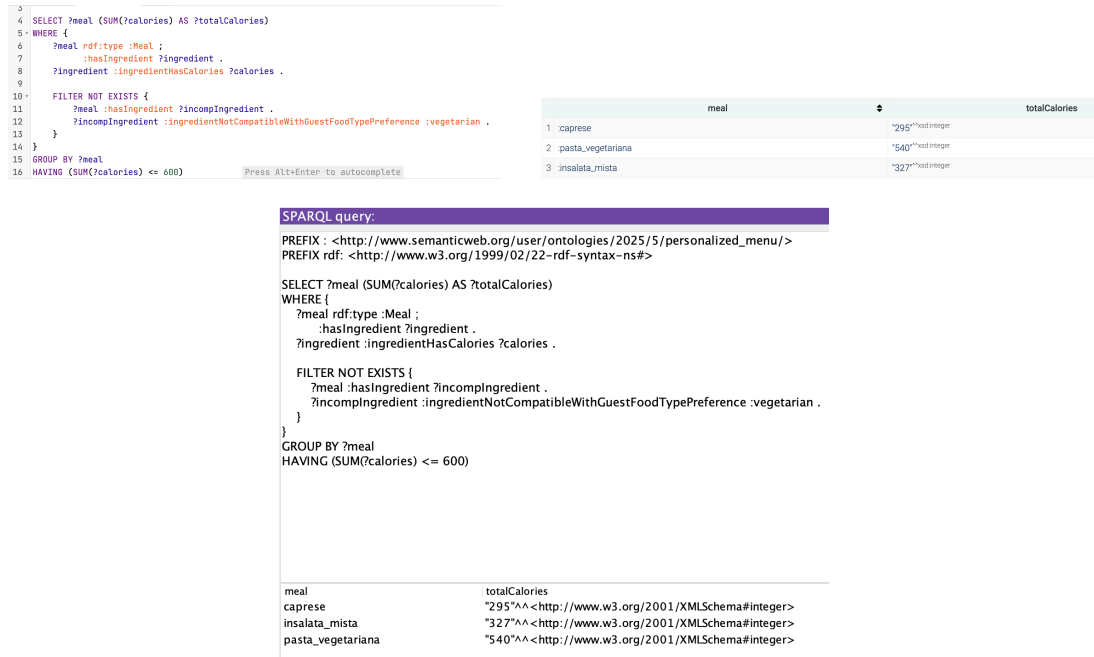


Figure 2.17: SPARQL query and results for calorie-conscious and vegetarian meals execution on GraphDB and Protégé.

### 2.3.4 SHACL Shape

To ensure data correctness and consistency, SHACL shapes (constraints for validating RDF data) were defined. These shapes verify that:

- Key classes such as `Guest`, `Meal`, `Ingredient`, and `Profile` have the correct types and appropriate cardinalities.
- Each `Ingredient` and `Meal` has exactly one calorie value, which must be an integer greater than or equal to 1.
- The meals are properly related to their ingredients, with at least one ingredient present.

```

1
2   :GuestShape a sh:NodeShape ;
3       sh:targetClass :Guest ;
4       sh:property [
5           sh:path :hasProfile ;
6           sh:class :GuestProfile ;
7       ] .

```

Listing 2.13: SHACL Shape for Guest

```

1
2   :IngredientShape a sh:NodeShape ;
3       sh:targetClass :Ingredient ;
4       sh:property [
5           sh:path :ingredientHasCalories ;
6           sh:datatype xsd:integer ;
7           sh:minInclusive 1 ;
8           sh:minCount 1 ;
9           sh:maxCount 1 ;
10      ] ;
11      sh:property [
12          sh:path :IngredientNotCompatibleWithFoodTypePreference ;
13          sh:class :GuestFoodTypePreference ;
14      ] .

```

Listing 2.14: SHACL Shape for Ingredient

```

1   :MealShape a sh:NodeShape ;
2       sh:targetClass :Meal ;
3       sh:property [
4           sh:path :mealHasCalories ;
5           sh:datatype xsd:integer ;
6           sh:minInclusive 1 ;
7           sh:minCount 1 ;
8           sh:maxCount 1 ;
9       ] ;
10      sh:property [
11          sh:path :hasIngredient ;
12          sh:class :Ingredient ;
13          sh:minCount 1 ;
14      ] .

```

Listing 2.15: SHACL Shape for Meal

By introducing SHACL validation, the system ensures that the data adheres to the expected schema and constraints before any reasoning or querying takes place. This validation layer reduces errors caused by incomplete or inconsistent data, enhancing the reliability and maintainability of the ontology.

## Validation

The SHACL validation was performed using the previously defined shapes. The results are shown in the figure below:

```
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix : <http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

# Shape for Guest
:GuestShape a sh:NodeShape ;
  sh:targetClass :Guest ;
  sh:property [
    sh:path :hasProfile ;
    sh:class :GuestProfile ;
  ] .

# Shape for GuestProfile
:GuestProfileShape a sh:NodeShape ;
  sh:targetClass :GuestProfile .

# Shape for GuestCaloriesPreference
:GuestCaloriesPreferenceShape a sh:NodeShape ;
  sh:targetClass :GuestCaloriesPreference .

# Shape for GuestFoodTypePreference
:GuestFoodTypePreferenceShape a sh:NodeShape ;
  sh:targetClass :GuestFoodTypePreference .

SHACL constraint violations: none
```

Figure 2.18: Result of SHACL validation

As shown in the image, no violations were found. This means that the data in the knowledge base conforms to all the specified SHACL constraints and is considered valid.

### 2.3.5 Knowledge Graphs

This solution adopts an ontology-based approach, which is more effective in representing complex relationships and meanings compared to other methods. By combining the ontology with SWRL rules and SPARQL queries, it enables both automatic reasoning (inferring new facts from existing data) and precise querying based on a structured and semantically rich model.

Another advantage of this approach is its extensibility. It can be easily connected to other vocabularies or external data sources, making the system scalable and adaptable to future needs.

Figures 2.19 and 2.20 show two examples of knowledge graphs created for guests with specific dietary restrictions: lactose intolerance and vegetarian. These graphs represent the connections between meals, ingredients, and guest preferences, and how they are used to filter suitable meal options.

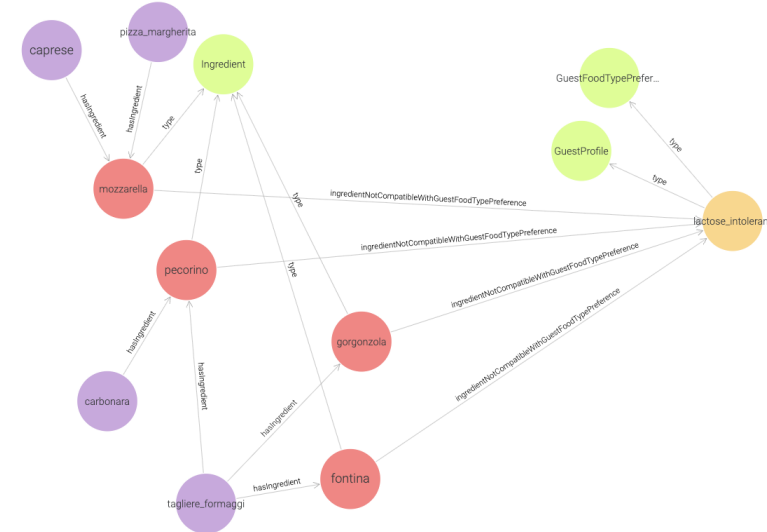


Figure 2.19: Knowledge graph for lactose-intolerant guests



Figure 2.20: Knowledge graph for vegetarian guests





## 3. Agile and Ontology-Based Meta-Modelling

In this section, we describe the agile and ontology-based approach used to extend a BPMN 2.0 process to recommend personalized meals based on guest preferences. The main goal was to connect the business process with the knowledge stored in an ontology and to use it to make intelligent, context-sensitive meal suggestions. This approach also helps visualize the reasoning process in a way that is easy for a restaurant manager to understand.

### 3.1 AOAME

**AOAME** (Agile Ontology-based Approach for Meta-Modelling Environment) is a prototype tool created to support the flexible and fast design of domain-specific modeling languages, especially useful for building Enterprise Knowledge Graphs (EKGs). These graphs are used to organize structured knowledge related to specific domains such as meals, ingredients, and guest preferences.

AOAME allows users to create, update, or hide modeling elements using metamodeling operators. These operators automatically generate SPARQL queries to update the triplestore, keeping the graphical model and the underlying ontology synchronized.

The tool is developed in Java and uses Apache Jena Fuseki to manage the ontology stored in the triplestore. It also provides APIs for interacting with the ontology and a graphical user interface to easily manage the modeling elements.

In this project, AOAME was used to extend BPMN 2.0 ontology-aware elements. This extension allows for the connection of process tasks with reasoning over guest profiles, enabling dynamic suggestions of personalized meals during process execution.

### 3.2 BPMN 2.0

**BPMN (Business Process Model and Notation)** is a standard graphical notation used to describe business processes in a simple and clear way. It helps to represent the sequence of activities and the flow of information within a process. BPMN is easy to understand for both technical and non-technical stakeholders.

The main components of BPMN 2.0 include the following:

- **Events:** Represent something that happens in the process. For example, a start event indicates when the process begins and an end event shows when it finishes.

- **Activities:** These are the tasks that need to be done, like filling in a form or selecting a meal. They can be done by a person (user task) or automatically by a system (service task).
- **Gateways:** Used to control the flow of the process. For example, a gateway can split the flow into different paths depending on a condition or join paths together.
- **Pools and Lanes:** These are used to show the participants involved in the process. A pool represents a main actor (such as the restaurant), and lanes divide the pool into specific roles (like customer and system).

In our project, we used BPMN to model the steps that a restaurant manager or customer might follow when ordering food. For example, the process includes a task in which the customer can say if they have allergies or follow a specific diet (like vegetarian, lactose-intolerant or gluten-intolerant). Then, the system filters the meals based on these preferences and shows only the suitable options. Finally, the customer can choose the meal that they want to order. This makes the ordering process safer and more personalized.

### 3.2.1 BPMN Process Description

The process starts when the guest scans a QR code with their smartphone to access the digital menu. After scanning, the guest sets their dietary preferences, such as vegetarian, gluten intolerant, lactose intolerant, or calorie conscious. These preferences are then sent to the recommendation system, which queries the ontology-based knowledge base to identify meals that match the specified requirements. The guest is presented with personalized meal suggestions and can choose a meal to place an order. Once the order is placed, the restaurant receives the request and the process ends.

Figure 3.1 shows the BPMN model representing this process.

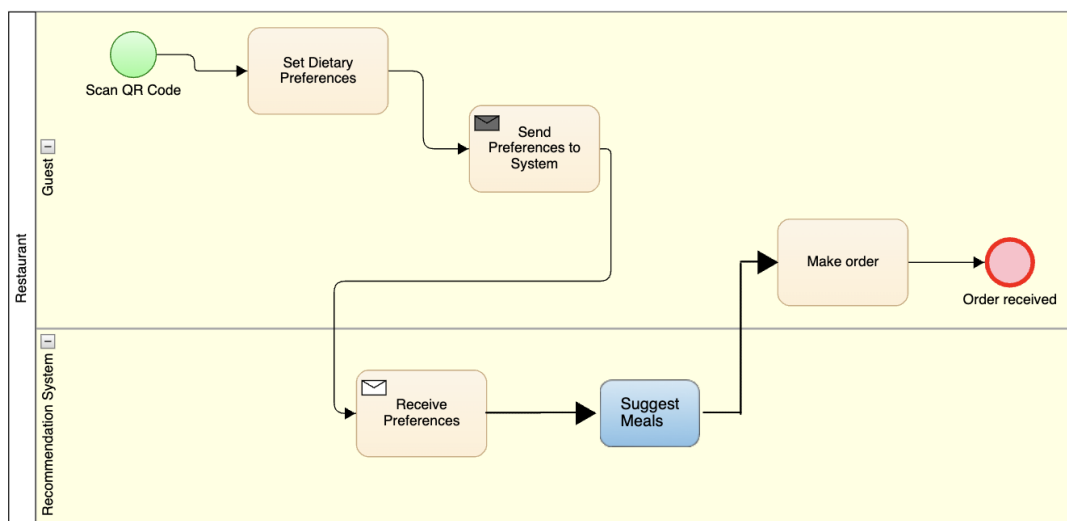


Figure 3.1: BPMN model of the personalized menu recommendation process

## Extending BPMN 2.0 for Meal Suggestion

To enable personalized meal recommendations within this workflow, the standard BPMN 2.0 *Task* element was extended by defining a new subclass called *Suggest Meals*. This specialized task adds domain-specific properties to the BPMN model, linking it directly to the ontology that stores information about meals, ingredients, and guest profiles.

Figure 3.2 shows the creation of the *Suggest Meals* task in the extended BPMN model.

The screenshot shows a web-based interface for creating a new sub-class of a BPMN Task. The title is 'Create new sub-class of Task'. There are three tabs at the top: 'New Modeling Element' (selected), 'Integrate with Existing Elements', and 'Add IoT Device'. The form contains the following fields:

- Parent Element:** Task
- Prefix \*:** bpmn:
- Child Element \*:** Suggest Meals (highlighted in blue)
- Comment:** A text input field.
- Category:** A dropdown menu with 'Activities' selected.
- Palette Image \*:** A file selection field with a 'Scegli file' button and a dropdown menu showing 'Task.png'.
- Canvas Image \*:** A file selection field with a 'Scegli file' button and a dropdown menu showing 'Task.png'.
- Buttons:** 'Cancel' and 'Create New Modeling Element' (in blue).

Figure 3.2: Creation of the *Suggest Meals* task in the BPMN model

The *Suggest Meals* task includes several boolean datatype properties to represent guest dietary preferences: *calorie\_conscious*, *lactose\_intolerant*, *gluten\_intolerant*, and *vegetarian*. These properties are shown in Figure 3.3.

The screenshot shows a dialog titled 'Edit Datatype Property'. At the top is a red button labeled 'Insert new Datatype Property'. Below it is a table with the following data:

Property Name	Range	Action
calorie_conscious	Range:xsd:boolean	▼
gluten_intolerant	Range:xsd:boolean	▼
lactose_intolerant	Range:xsd:boolean	▼
vegetarian	Range:xsd:boolean	▼

At the bottom left is a 'Cancel' button.

Figure 3.3: Guest preference properties for the *Suggest Meals* task

When the *Suggest Meals* task is executed during the process, it triggers a SPARQL query to the ontology stored in the triple store (Apache Jena Fuseki). This query filters the available meals based on the guest's dietary preferences and restrictions, returning only those meals that are compatible. This dynamic filtering allows the system to present personalized meal options directly within the business process flow.

### 3.2.2 SPARQL Query Execution

After modeling the BPMN 2.0 process, we used Apache Jena Fuseki to execute SPARQL queries on the ontology.

Apache Jena Fuseki is a scalable SPARQL server that efficiently manages RDF datasets. It provides a RESTful interface to perform operations such as executing SPARQL queries for data retrieval, performing SPARQL updates to modify data, and managing RDF datasets.

The semantic execution of the extended BPMN process relies on SPARQL queries that dynamically filter available meals based on the guest's dietary preferences and restrictions. This integration enables the business process to make personalized and context-aware decisions by querying the ontology at runtime.

### SPARQL Query for Personalized Meal Suggestion

To implement personalized meal recommendations, the *Suggest Meals* task executes the following SPARQL query, which filters meals according to the guest's preferences:

```

3 PREFIX mod: <http://fhnw.ch/modelingEnvironment/ModelOntology#>
4 PREFIX lo: <http://fhnw.ch/modelingEnvironment/LanguageOntology#>
5 PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
6 PREFIX : <http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/>
7 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
8
9 SELECT DISTINCT ?meal
10 WHERE {
11   {
12     SELECT ?isVegetarian ?isLactoseIntolerant ?isGlutenIntolerant ?isCalorieConscious WHERE {
13       mod:SuggestMeals_aa7a29e3-22db-4c23-a5d7-d18eabf04d92 lo:vegetarian ?vegetarianObj .
14       mod:SuggestMeals_aa7a29e3-22db-4c23-a5d7-d18eabf04d92 lo:lactose_intolerant ?lactoseObj .
15       mod:SuggestMeals_aa7a29e3-22db-4c23-a5d7-d18eabf04d92 lo:gluten_intolerant ?glutenObj .
16       mod:SuggestMeals_aa7a29e3-22db-4c23-a5d7-d18eabf04d92 lo:calorie_conscious ?calorieObj .
17       BIND(xsd:boolean(?vegetarianObj) AS ?isVegetarian)
18       BIND(xsd:boolean(?lactoseObj) AS ?isLactoseIntolerant)
19       BIND(xsd:boolean(?glutenObj) AS ?isGlutenIntolerant)
20       BIND(xsd:boolean(?calorieObj) AS ?isCalorieConscious)
21     }
22
23     ?meal rdf:type :Meal .
24
25     FILTER(
26       (?isVegetarian = false || NOT EXISTS {
27         ?meal :hasIngredient ?ingredient .
28         ?ingredient :ingredientNotCompatibleWithFoodTypePreference :vegetarian .
29       }) &&
30       (?isLactoseIntolerant = false || NOT EXISTS {
31         ?meal :hasIngredient ?ingredient .
32         ?ingredient :ingredientNotCompatibleWithFoodTypePreference :lactose_intolerant .
33       }) &&
34       (?isGlutenIntolerant = false || NOT EXISTS {
35         ?meal :hasIngredient ?ingredient .
36         ?ingredient :ingredientNotCompatibleWithFoodTypePreference :gluten_intolerant . })
37
38     FILTER(
39       (?isCalorieConscious = false || EXISTS {
40         SELECT ?meal (SUM(?calories) AS ?totalCalories)
41         WHERE {
42           ?meal :hasIngredient ?ingredient .
43           ?ingredient :ingredientHasCalories ?calories .
44         }
45         GROUP BY ?meal
46         HAVING (SUM(?calories) <= 600)
47       })
48   )
49 }
```

Figure 3.4: SPARQL query executed by the 'Suggest Meals' task in Jena Fuseki

- The query first extracts the boolean values representing the guest's dietary preferences (vegetarian, lactose intolerant, gluten intolerant, and calorie conscious) from the ontology.
- It then selects all meals defined in the ontology.
- For each dietary restriction set to *true*, the query excludes meals that contain incompatible ingredients by checking the property `:ingredientNotCompatibleWithFoodTypePreference`.
- For the calorie-conscious preference, it sums the calories of all ingredients in each meal and only includes meals with a total calorie count less or equal to 600.

This approach enables dynamic, fine-grained filtering of meals in real time, ensuring that only suitable options are presented to the guest based on their preferences.

## Results Overview

The query results dynamically change based on the selected input preferences. For example, by enabling only the *vegetarian* option, the system filters out all meals containing ingredients incompatible with vegetarian diets. Below are two example scenarios:

### Model element attributes

ID: SuggestMeals\_aa7a29e3-22db-4c23-a5d7-d18eabf04d92

Instantiation Type: Instance

Relation	Value	Actions
vegetarian	<input type="text" value="true"/>	<button>Remove</button>
lactose_intolerant	<input type="text" value="false"/>	<button>Remove</button>
gluten_intolerant	<input type="text" value="false"/>	<button>Remove</button>
calorie_conscious	<input type="text" value="false"/>	<button>Remove</button>

Figure 3.5: Configuration of the *vegetarian* attribute in the *Suggest Meals* extended task

meal	
1	<a href="http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/caprese">http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/caprese</a>
2	<a href="http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/insalata_mista">http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/insalata_mista</a>
3	<a href="http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/pasta_vegetariana">http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/pasta_vegetariana</a>
4	<a href="http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/tagliere_formaggi">http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/tagliere_formaggi</a>
5	<a href="http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/pizza_margherita">http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/pizza_margherita</a>

Figure 3.6: SPARQL query result showing meals compatible with a vegetarian preference

In the first example, as shown in the figures above, we configured the *vegetarian* attribute in the extended BPMN task. Once the SPARQL query was executed in Apache Jena Fuseki, the system returned a list of meals compatible with the vegetarian constraint.

## Model element attributes

ID: SuggestMeals\_aa7a29e3-22db-4c23-a5d7-d18eabf04d92

Instantiation Type: Instance

Relation	Value	Actions
vegetarian	<input type="text" value="false"/>	<button>Remove</button>
lactose_intolerant	<input type="text" value="false"/>	<button>Remove</button>
gluten_intolerant	<input type="text" value="true"/>	<button>Remove</button>
calorie_conscious	<input type="text" value="false"/>	<button>Remove</button>

Figure 3.7: Configuration of the *gluten\_intolerant* attribute in the *Suggest Meals* extended task

meal	
1	<a href="http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/caprese">http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/caprese</a>
2	<a href="http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/insalata_mista">http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/insalata_mista</a>
3	<a href="http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/pollo_grigliato">http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/pollo_grigliato</a>
4	<a href="http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/risotto_gamberetti">http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/risotto_gamberetti</a>
5	<a href="http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/tagliata_di_manzo">http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/tagliata_di_manzo</a>
6	<a href="http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/tagliere_formaggi">http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/tagliere_formaggi</a>
7	<a href="http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/tagliere_salumi">http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/tagliere_salumi</a>

Figure 3.8: SPARQL query result showing meals compatible with a gluten intolerant preference

In the second example, we enabled the *gluten\_intolerant* option. The query execution returned only the meals that do not contain any gluten-incompatible ingredients, demonstrating how dietary filters are correctly applied in real time.



## Model element attributes

ID: SuggestMeals\_aa7a29e3-22db-4c23-a5d7-d18eabf04d92

Instantiation Type: Instance

Relation	Value	Actions
vegetarian	<input type="text" value="false"/>	<button>Remove</button>
lactose_intolerant	<input type="text" value="false"/>	<button>Remove</button>
gluten_intolerant	<input type="text" value="false"/>	<button>Remove</button>
calorie_conscious	<input type="text" value="true"/>	<button>Remove</button>

Figure 3.9: Configuration of the *calorie\_conscious* attribute in the *Suggest Meals* extended task

meal	
1	<a href="http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/caprese">http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/caprese</a>
2	<a href="http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/insalata_mista">http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/insalata_mista</a>
3	<a href="http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/pasta_vegetariana">http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/pasta_vegetariana</a>
4	<a href="http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/pollo_grigliato">http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/pollo_grigliato</a>
5	<a href="http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/risotto_gamberetti">http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/risotto_gamberetti</a>
6	<a href="http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/tagliata_di_manzo">http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/tagliata_di_manzo</a>

Figure 3.10: SPARQL query result showing meals compatible with a calorie conscious preference

In this third example, the *calorie\_conscious* option was enabled. The system returned only meals whose total calorie content does not exceed 600 kilocalories, as defined by the SPARQL query logic.

## Model element attributes

ID: SuggestMeals\_aa7a29e3-22db-4c23-a5d7-d18eabf04d92

Instantiation Type: Instance

Relation	Value	Actions
vegetarian	<input type="text" value="true"/>	<button>Remove</button>
lactose_intolerant	<input type="text" value="false"/>	<button>Remove</button>
gluten_intolerant	<input type="text" value="false"/>	<button>Remove</button>
calorie_conscious	<input type="text" value="true"/>	<button>Remove</button>

Figure 3.11: Configuration of the *vegetarian* and *calorie\_conscious* attributes in the *Suggest Meals* extended task

meal	
1	<a href="http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/caprese">http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/caprese</a>
2	<a href="http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/insalata_mista">http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/insalata_mista</a>
3	<a href="http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/pasta_vegetariana">http://www.semanticweb.org/user/ontologies/2025/5/personalized_menu/pasta_vegetariana</a>

Figure 3.12: SPARQL query result showing meals compatible with both vegetarian and calorie conscious preferences

For this final example, we enabled both the *vegetarian* and *calorie\_conscious* preferences in the extended BPMN task. The SPARQL query executed in Apache Jena Fuseki returned only those meals that are free of ingredients incompatible with vegetarian diets and have a total calorie count equal to or less than 600. This demonstrates how multiple dietary constraints can be combined to refine the meal suggestions with high precision.



## 4. Conclusions

This project was a good opportunity to explore different ways of representing and using knowledge to support personalized menu suggestions. We followed a step-by-step approach, starting with decision logic using DMN, then moving to logic programming with Prolog, and finally creating a semantic solution with ontologies, SWRL rules, SPARQL queries, and SHACL validation.

Each solution had its own strengths. DMN was simple and easy to understand, useful for basic decisions. Prolog allowed us to write more complex rules and manage logical relationships. The ontology-based approach gave us more advanced features like inference, data validation, and the possibility to run flexible queries on the knowledge base.

In the last part of the project, we used this semantic knowledge inside a BPMN 2.0 process model. Thanks to the AOAME framework and the Jena Fuseki interface, we were able to connect BPMN tasks with the knowledge graph. This made the process smarter and able to change based on guest preferences and other conditions.

Overall, this combination of semantic technologies and process modeling showed how it's possible to build systems that can adapt to different situations and support better decision-making, especially in areas like food service where personalization is important.

### 4.1 Elia Toma - Conclusions

The conclusions of this work reflect a rich and multi-disciplinary experience across the diverse fields of rule-based reasoning, logic programming, ontology engineering, and semantic meta-modelling. By engaging with multiple technologies and approaches, I have gained valuable insights into the strengths and limitations of each paradigm, as well as the complementary power they offer when combined in a cohesive solution.

- **Decision Tables:** This technique provided a clear and structured method to model business rules. Its graphical representation allowed for intuitive specification of meal filtering logic based on user preferences. Despite its limited expressiveness, decision tables were effective for encoding straightforward conditions, and Camunda proved to be a valuable tool for simulation and testing.
- **Prolog:** Through Prolog, I explored the expressiveness of logic programming for modeling guest preferences and meal suggestions. The use of rules and backtracking allowed us to define complex conditions, especially recursive calorie calculations and dietary filtering. Although it lacks a formal semantic model, Prolog demonstrated flexibility and simplicity for rule-based reasoning.
- **Knowledge Graph/Ontology:** Designing the ontology in Protege involved the formal specification of all relevant classes, properties, and relationships between

concepts such as Guest, Meal, and Ingredient. By defining SWRL rules, SPARQL queries, and SHACL constraints, I enabled inferencing, data validation, and advanced querying capabilities. The ontology served as a centralized and semantically rich representation of the domain.

- **AOAME:** In the final task, I experimented with AOAME to extend BPMN 2.0 semantics. The ontology created in Protege was imported in Jena Fuseki, and a custom SPARQL query was written and executed from AOAME to retrieve meals matching user-defined constraints. This integration showed how business processes can leverage semantic data to become context-aware and responsive.

In conclusion, this project has deepened my understanding of how symbolic AI and business modeling frameworks can be harmonized. Each layer of the solution contributed to a system capable of not only recommending meals based on constraints, but also adapting its behavior within a modeled process. I believe the combination of rule-based reasoning, ontological representation, and semantic-aware meta-modelling offers a promising approach for intelligent systems design, applicable far beyond the food domain.

## 4.2 Sofia Scattolini - Conclusions

This project was a good opportunity to explore different knowledge-based approaches in a practical and structured way. Together with my colleague, I worked on the knowledge engineering part of the system, using various technologies to support personalized menu recommendations. Each solution had its strengths and limitations depending on how the logic was structured and how flexible or complex it needed to be. Below, I reflect on the different approaches we used and what I learned from each of them.

**Decision Tables** I found decision tables easy to understand and quick to use, especially for rules related to food preferences like allergies or diets. Using DRDs helped me visualize how the decisions were connected. At the same time, I noticed that when the number of conditions increased, the tables became difficult to manage. Personally, I think this approach is good for simple rules but less flexible when the logic becomes more complex.

**Logical Reasoning with Prolog** Prolog allowed me to express rules and relationships between guests, meals, and ingredients in a clear, logical way. It was useful to model complex constraints and automate reasoning based on facts. However, its syntax is less common among traditional programming languages, which can make it less intuitive for some users. Additionally, as the number of rules grows, organizing and maintaining them becomes increasingly complex and difficult to scale.

**Ontology-Based Approach** The ontology-based solution provided a formal and semantic representation of meals, ingredients, and guest preferences. Using OWL, SWRL, SPARQL, and SHACL allowed the system to infer new knowledge, validate data, and perform complex queries over the knowledge graph. This approach supports interoperability and advanced reasoning, but it requires a solid understanding of ontology design principles and reasoning mechanisms.

**AOAME and BPMN Integration** This part was especially interesting for me because I had already studied BPMN in some of my courses. With AOAME, I saw how it is possible to extend standard BPMN by embedding semantic knowledge into the tasks. It was rewarding to see how the process model could “understand” which meal fits each guest, thanks to the ontology. I worked on creating the semantic version of the task and helped define how it connects to the knowledge base. I believe this integration could be very useful in real-world applications, where decisions depend on user preferences or dynamic data.

**Final Thoughts** Working on this project allowed me to apply and combine concepts from my studies, bridging technical knowledge with business process understanding. Exploring these different approaches gave me a clearer view of how intelligent, knowledge-driven, and context-aware systems can be designed and implemented. This experience strengthened my skills and confidence in using knowledge engineering tools and methodologies, which I believe will be valuable for future projects in data-driven and process-oriented domains.



# References

- [1] Elia Toma and Sofia Scattolini. KEBI-Personalized\_Menu. [https://github.com/Elia-Toma/KEBI-personalized\\_menu](https://github.com/Elia-Toma/KEBI-personalized_menu), 2025.
- [2] Trisotech. Decision model and notation (dmn) modeler, 2025. URL <https://www.trisotech.com/dmn/>.
- [3] Jan Wielemaker et al. Swish - swi-prolog for sharing. <http://swish.swi-prolog.org/>, 2025.
- [4] Stanford Center for Biomedical Informatics Research. Protégé ontology editor and knowledge acquisition system. <https://protege.stanford.edu/>, 2025.
- [5] Ontotext. Graphdb semantic graph database. <https://www.ontotext.com/products/graphdb/>, 2025.