Personalized Menu Project

MSc Computer Science, Knowledge Engineering course

Fabio Grelloni Mat.132549

Claudio Cozzolino Mat.132015

A.Y. 2024/25

Abstract. This project aims to design a knowledge-based recommendation system for digital restaurant menus. By leveraging knowledge representation techniques—including decision tables, logic programming, ontologies, and meta-modelling. This system dynamically adapts to different guest profiles such as vegetarians, vegans, carnivores, calorie-conscious individuals, and those with allergies. The project also introduces an ontology-driven BPMN adaptation to model the suggestion process using an agile and explainable approach. The overall goal is to enhance the digital menu experience, improving accessibility, relevance, and satisfaction for different guests.

Contents

Listings

List of Figures

1 Introduction

In recent years, the digitalization of restaurant menus has become increasingly common, with many establishments adopting QR code systems to present their menus directly on guest's smartphones. While this provides convenience and reduces the need for physical menus, it also introduces limitations, particularly the small screen size, which can make it difficult to browse large or complex menus efficiently. This challenge becomes even more significant for guests with specific dietary preferences or restrictions, such as vegetarians, calorie-conscious individuals, or those with allergies like lactose or gluten intolerance.

To improve the dining experience for such guests, a more intelligent and personalized approach to menu presentation is needed. Instead of displaying all available meals, it is preferable to show only those options that align with each guest's individual needs and preferences. This project, "Personalized Menu," addresses this challenge by leveraging knowledge-based technologies to filter and recommend suitable meals, ensuring that each guest can make informed choices quickly and comfortably.

The project integrates various knowledge representation and reasoning techniques to model both the restaurant's meal offerings and the diverse profiles of guests. In doing so, it creates a system capable of dynamically recommending meals based on preferences such as dietary type, calorie sensitivity, or specific allergies, enhancing both accessibility and user satisfaction.

2 Project Objective

The objective of the "Personalized Menu" project is to design and implement a knowledge-based system that intelligently recommends meals to restaurant guests based on their dietary preferences, nutritional concerns, and potential food allergies.

To achieve this, the project will:

- Develop a comprehensive knowledge base representing typical meals found in Italian restaurants, including pizzas, pasta dishes, and main courses, along with detailed ingredient information such as food categories (meat, vegetables, dairy, etc.) and calorie content.
- Model guest profiles, considering dietary preferences (e.g., carnivore, vegetarian), calorie-conscious behavior, and common food intolerances or allergies (e.g., lactose, gluten).
- Implement multiple knowledge-based solutions for personalized meal recommendations using:
  - **Decision Tables and Decision Requirement Diagrams (DRD)**, created with *Camunda Decision Modeler*, to represent structured decision logic in an intuitive, visual format.
  - **Prolog**, developed and tested with *SWISH Prolog*, for logic programming with facts and rules that define meal compatibility.
  - **Knowledge Graphs and Ontologies**, modeled in *Protégé*, enriched with SWRL rules, SPARQL queries, and SHACL constraints to enable semantic reasoning and rule-based filtering of meals.
- Extend the solution with **ontology-based meta-modelling** using *AOAME*, adapting BPMN 2.0 to visually support meal recommendations. The BPMN extension introduces new modeling elements tailored for the restaurant context, providing an accessible graphical interface for restaurant managers.
- Integrate the system with a semantic infrastructure using *Jena Fuseki* as a triple store to execute SPARQL queries and dynamically retrieve meals compatible with guest preferences.

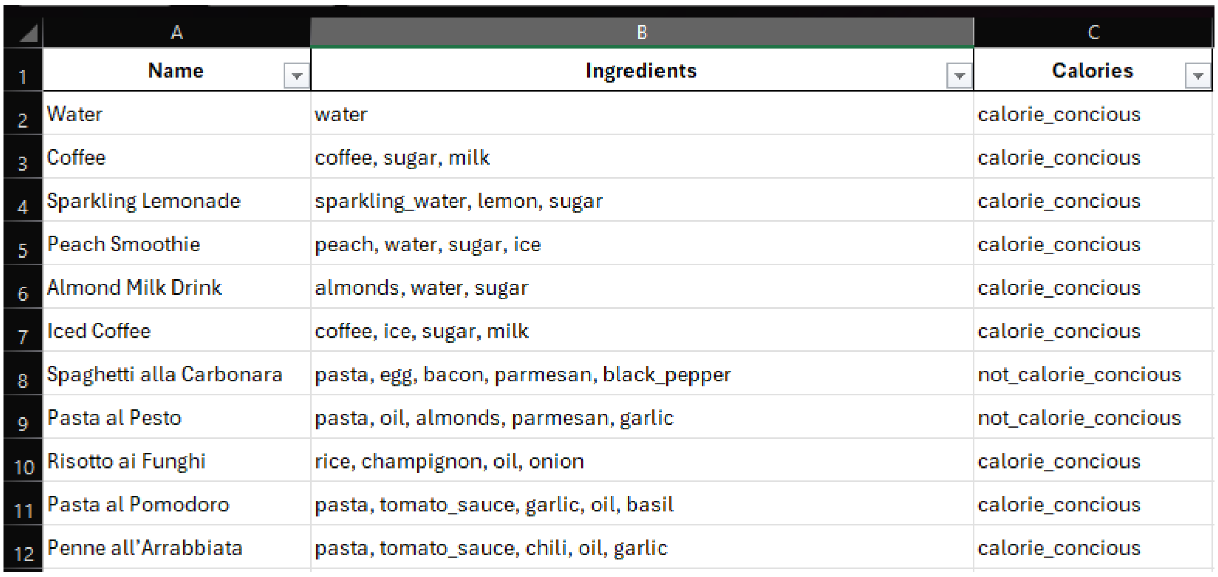
All related files, implementation details, and project documentation are publicly accessible on GitHub at the following link: `https://github.com/HertzKaa/Kebi_Exam_24-25`.

By comparing these different knowledge-based approaches, the project evaluates their respective advantages and limitations, offering insights into effective strategies for building intelligent, personalized recommendation systems within the restaurant domain.

3 Menu Structure

As part of the "Personalized Menu" project, a complete menu has been created, consisting of **42 items**, reflecting typical dishes of an Italian restaurant. Each menu item is described with multiple attributes: **Name**, **Ingredients**, **Calories**. Another table has been created that lists the ingredients of each dish. Each ingredient has a **Diet**, **Dairy**, **Eggs**, **Gluten**, **Nuts**, **Shellfish** field.

With this number of dishes and the wide range of allergens and dietary options considered, the created menu is as comprehensive as possible and very close to a realistic restaurant scenario. It covers the most common allergens and dietary restrictions encountered in real-world dining experiences, making it suitable for testing advanced, practical recommendation systems.



	A	B	C
1	Name	Ingredients	Calories
2	Water	water	calorie_concious
3	Coffee	coffee, sugar, milk	calorie_concious
4	Sparkling Lemonade	sparkling_water, lemon, sugar	calorie_concious
5	Peach Smoothie	peach, water, sugar, ice	calorie_concious
6	Almond Milk Drink	almonds, water, sugar	calorie_concious
7	Iced Coffee	coffee, ice, sugar, milk	calorie_concious
8	Spaghetti alla Carbonara	pasta, egg, bacon, parmesan, black_pepper	not_calorie_concious
9	Pasta al Pesto	pasta, oil, almonds, parmesan, garlic	not_calorie_concious
10	Risotto ai Funghi	rice, champignon, oil, onion	calorie_concious
11	Pasta al Pomodoro	pasta, tomato_sauce, garlic, oil, basil	calorie_concious
12	Penne all'Arrabbiata	pasta, tomato_sauce, chili, oil, garlic	calorie_concious

Figure 1: Example of meals with ingredients

4 Decision Model: Camunda

Camunda allows for modeling business decisions using the **DMN language**, a standard that provides a visual and formal representation of decision rules. This approach helps to define decision logic clearly and understandably, enabling companies to create models that reflect their decision-making policies and procedures. Camunda uses decision diagrams to represent decisions and decision rules. The diagrams can include decision tables, which display rules and criteria in a tabular format, making it easier to understand and manage complex decisions. Once modeled, decisions can be executed directly within the Camunda platform. Camunda integrates decision rules into business processes and automates them, enhancing the efficiency and consistency of decisions.

4.1 Decision Modeler

The Figure 2 shows our decision model, which takes as input the list of available ingredients and the list of all dishes included in the menu. Additionally, the model incorporates the customer's dietary preferences, such as lactose intolerance, gluten intolerance, nuts allergy, eggs allergy, shellfish allergy, vegetarian choice, vegan choice and the desired calorie level. We have modified the definitions of the rules within the decision tables to ensure that the execution is less script-like and more aligned with the decision-making paradigm.

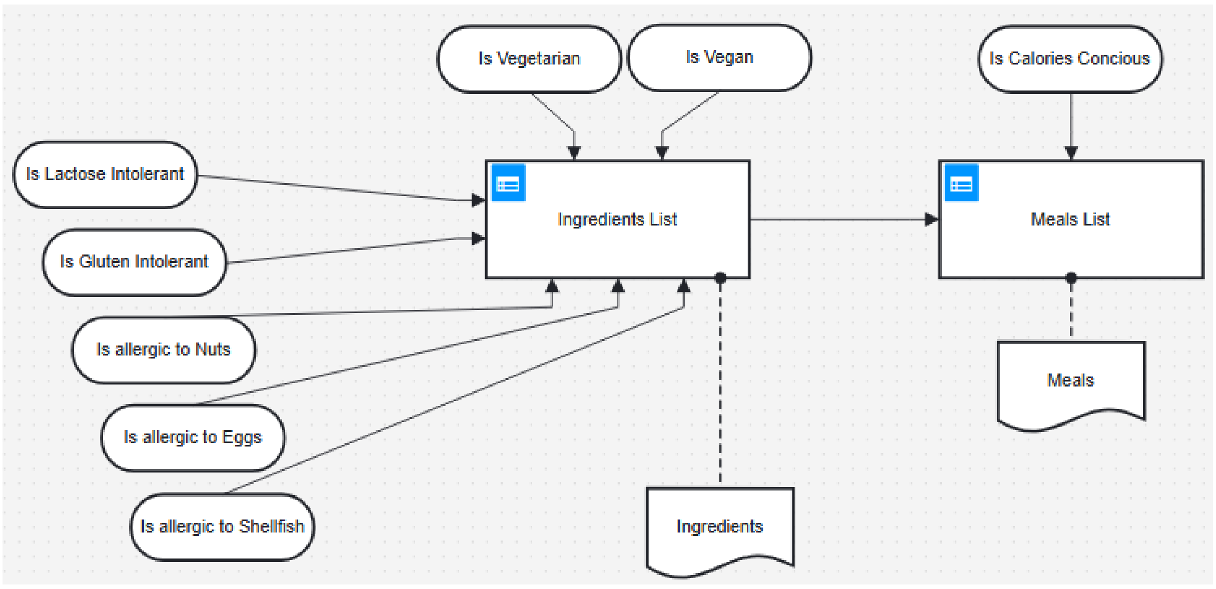


Figure 2: Overall structure of the Decision Model.

4.2 Elements of the Decision Diagram

4.2.1 Knowledge Sources

Knowledge Sources represent the fundamental sources of knowledge that provide the data and information necessary to support the decision-making process. They are considered fixed resources that offer a knowledge base on which to apply decision rules and determine the most appropriate solution based on specific inputs. In our diagram, there are two Knowledge Sources: Ingredients and Meals (Figure 3). Ingredients represents the list of available ingredients in the restaurant, describing the ingredients at a theoretical level, while Meals represents the list of all the dishes available in the restaurant.

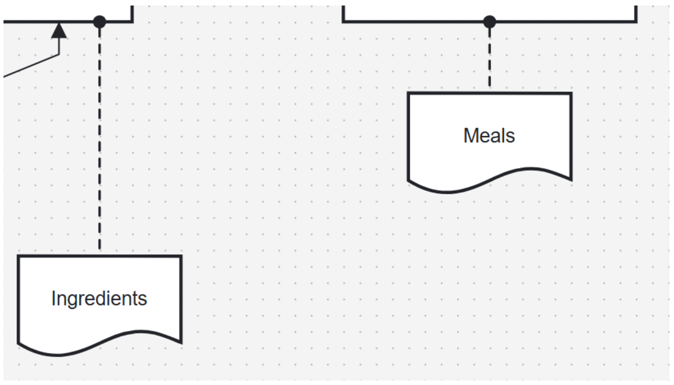


Figure 3: Knowledge Sources: Ingredients and Meals.

4.2.2 Input Data

Input Data represent the data provided to the system as input for the decision-making process. These data are typically variables that the user or another system provides during the execution of the process. In DMN, Input Data are used by the Decision Tables to calculate or determine the output. In our diagram, there are seven Input Data used to allow the customer to input possible intolerance and dietary preferences: is Lactose intolerant, Is Gluten Intolerant, Is allergic to Nuts, Is allergic to Eggs, Is allergic to Shellfish, Is Vegetarian, Is Vegan. These Input Data are of a boolean type. Therefore, if a customer is lactose intolerant, they must enter true, otherwise false. The same logic applies to gluten intolerance and vegetarian preferences. Calories level allows the customer to specify the preferred calorie level. If not specified, all dishes with any available calorie levels will be considered.

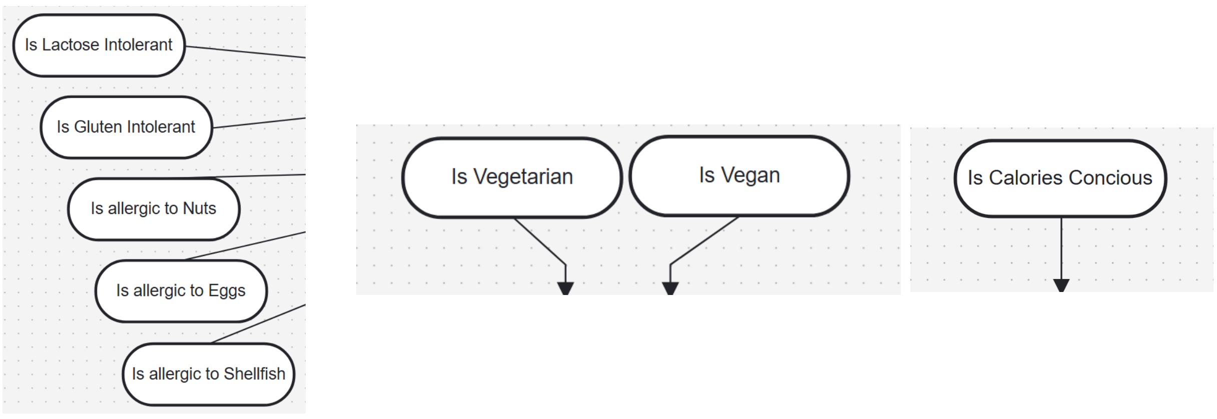
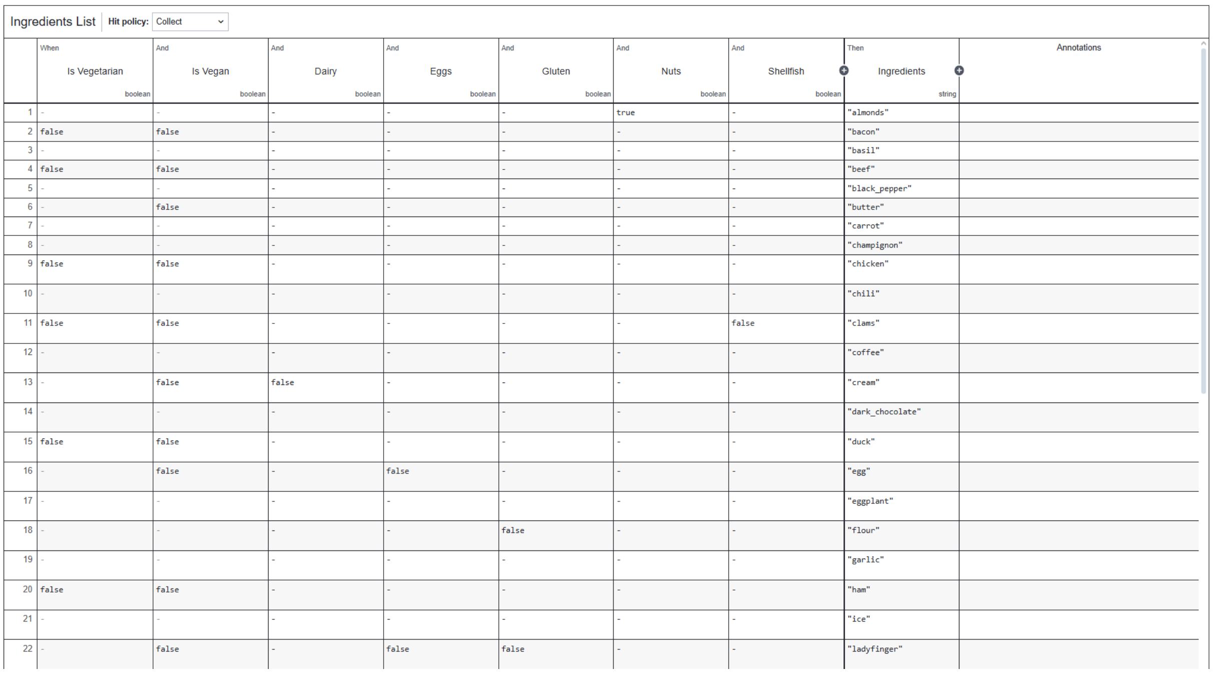


Figure 4: Inputs of the customer

4.2.3 Decision Tables

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

- Ingredients List: It aims to identify the ingredients that should not be present in the dishes, based on the customer's preferences. Some rules use the symbol "-" in the conditions, indicating that the specific condition is irrelevant for that rule. For example, a rule with "-" in all conditions means that the corresponding ingredient is selected regardless of any intolerance or diet. Conversely, if an ingredient is marked as false for one of the inputs, that ingredient will be excluded, and dishes containing it will not be available to the customer. The Collect hit policy indicates that if multiple rules are satisfied simultaneously, the corresponding results will be aggregated into a list. In the context of Selected ingredients, this means that various ingredients can be selected based on the specified conditions.

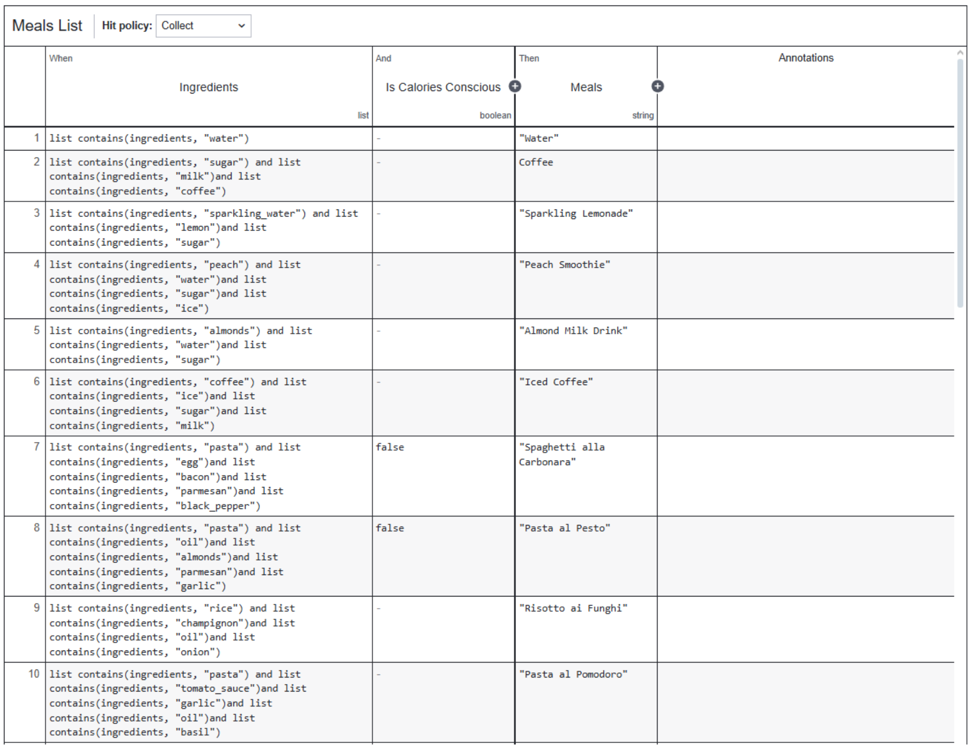


Ingredients List | Hit policy: Collect

	When	And	And	And	And	And	And	Then	
	Is Vegetarian	Is Vegan	Dairy	Eggs	Gluten	Nuts	Shellfish	Ingredients	Annotations
	boolean	boolean	boolean	boolean	boolean	boolean	boolean	string	
1	-	-	-	-	-	true	-	"almonds"	
2	false	false	-	-	-	-	-	"bacon"	
3	-	-	-	-	-	-	-	"basil"	
4	false	false	-	-	-	-	-	"beef"	
5	-	-	-	-	-	-	-	"black_pepper"	
6	-	false	-	-	-	-	-	"butter"	
7	-	-	-	-	-	-	-	"carrot"	
8	-	-	-	-	-	-	-	"champignon"	
9	false	false	-	-	-	-	-	"chicken"	
10	-	-	-	-	-	-	-	"chili"	
11	false	false	-	-	-	-	false	"clams"	
12	-	-	-	-	-	-	-	"coffee"	
13	-	false	false	-	-	-	-	"cream"	
14	-	-	-	-	-	-	-	"dark_chocolate"	
15	false	false	-	-	-	-	-	"duck"	
16	-	false	-	false	-	-	-	"egg"	
17	-	-	-	-	-	-	-	"eggplant"	
18	-	-	-	-	false	-	-	"flour"	
19	-	-	-	-	-	-	-	"garlic"	
20	false	false	-	-	-	-	-	"ham"	
21	-	-	-	-	-	-	-	"ice"	
22	-	false	-	false	false	-	-	"ladyfinger"	

Figure 5: Ingredient List.

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



Meals List | Hit policy: Collect

	When	And	Then	
	Ingredients list	Is Calories Conscious boolean	Meals string	Annotations
1	list contains(ingredients, "water")	-	"Water"	
2	list contains(ingredients, "sugar") and list contains(ingredients, "milk")and list contains(ingredients, "coffee")	-	Coffee	
3	list contains(ingredients, "sparkling_water") and list contains(ingredients, "lemon")and list contains(ingredients, "sugar")	-	"Sparkling Lemonade"	
4	list contains(ingredients, "peach") and list contains(ingredients, "water")and list contains(ingredients, "sugar")and list contains(ingredients, "ice")	-	"Peach Smoothie"	
5	list contains(ingredients, "almonds") and list contains(ingredients, "water")and list contains(ingredients, "sugar")	-	"Almond Milk Drink"	
6	list contains(ingredients, "coffee") and list contains(ingredients, "ice")and list contains(ingredients, "sugar")and list contains(ingredients, "milk")	-	"Iced Coffee"	
7	list contains(ingredients, "pasta") and list contains(ingredients, "egg")and list contains(ingredients, "bacon")and list contains(ingredients, "parmesan")and list contains(ingredients, "black_pepper")	false	"Spaghetti alla Carbonara"	
8	list contains(ingredients, "pasta") and list contains(ingredients, "oil")and list contains(ingredients, "almonds")and list contains(ingredients, "parmesan")and list contains(ingredients, "garlic")	false	"Pasta al Pesto"	
9	list contains(ingredients, "rice") and list contains(ingredients, "champignon")and list contains(ingredients, "oil")and list contains(ingredients, "onion")	-	"Risotto ai Funghi"	
10	list contains(ingredients, "pasta") and list contains(ingredients, "tomato_sauce")and list contains(ingredients, "garlic")and list contains(ingredients, "oil")and list contains(ingredients, "basil")	-	"Pasta al Pomodoro"	

Figure 6: Meal List.

4.2.4 Conclusion

In conclusion, the DMN diagrams and their underlying decision logic performed as intended, delivering accurate, realistic, and well-filtered meal recommendations, fully aligned with the project's objectives.

5 SWISH Prolog

Prolog (short for *Programming in Logic*) is a high-level programming language rooted in formal logic, specifically designed for tasks involving reasoning, rule-based decision making, and knowledge representation. Prolog is based on the paradigm of **declarative programming**, where developers define *what* the system should accomplish through facts and logical rules, rather than specifying *how* to achieve it step by step, as in traditional procedural languages.

At its core, Prolog operates on the principles of:

- **Facts:** Statements that represent known information or relationships within the system (e.g., `meal(pizza).` or `ingredient_info(flour, gluten).`).
- **Rules:** Logical relationships that infer new knowledge from existing facts (e.g., defining when a meal is suitable for vegetarians).
- **Queries:** Questions asked to the Prolog system, allowing users to retrieve information or verify conditions based on the existing knowledge base.

Prolog uses **backtracking** and **unification** to explore possible solutions when answering queries. This makes it particularly effective for tasks such as:

- Filtering data based on complex rules.
- Defining relationships between entities (e.g., ingredients, meals, dietary restrictions).
- Building expert systems and intelligent recommendation engines.

5.1 Prolog in the Personalized Menu Project

In the context of the "Personalized Menu" project, Prolog was used to:

- Define the complete knowledge base of meals, ingredients, dietary properties, and allergens through structured facts.
- Implement logical rules to determine meal compatibility with guest preferences, such as:
  - Filtering meals based on allergens.
  - Classifying meals according to dietary types (e.g., Vegan, Vegetarian, Carnivore).
  - Considering calorie-conscious preferences.
- Perform queries to dynamically retrieve meals that satisfy the given constraints.

The **SWISH online platform** was utilized for implementing and testing the Prolog logic. Thanks to Prolog's logic-driven structure, the system provides a highly flexible and accurate method of filtering meals, closely aligned with real-world reasoning processes.

Prolog's declarative approach makes it especially powerful for knowledge-based applications like this, where the relationships between data points and rules are complex, but the desired output (personalized meal recommendations) can be precisely defined.

5.2 Facts Definition in Prolog

The foundation of the knowledge base for the "Personalized Menu" project in **Prolog** is built upon a structured set of **facts**, which explicitly define the available meals and the info of every ingredient. These facts allow the system to reason about meal compatibility, filtering options, and personalized recommendations for different guest profiles.

5.2.1 Meal Declaration

The first set of facts declares the existence of each meal in the menu using the following format:

```
meal("Name", [ingredient/s], calories)
```

Listing 1: Meal declaration

For example:

```
meal("Chicken with Rice", [chicken, rice, peas, oil, salt],
    calorie_concious).
meal("Baked Sea Bass", [seabass, lemon, oil, salt, rosemary],
    calorie_concious).
meal("Sausage with Peas", [sausage, peas, onion, oil, salt],
    calorie_concious).
```

Listing 2: Example of meal declaration

5.2.2 Attributes of each ingredient

Each meal has its own ingredients, and each ingredient has its own attributes, namely its nutritional value and allergens:

```
ingredient_info(Name, diet, dairy, eggs, gluten, nuts, shellfish).
```

Listing 3: Ingredients info

Example:

```
ingredient_info(almonds, vegan, dairy_free, eggs_free, gluten_free, nuts
    , shellfish_free).
ingredient_info(bacon, carnivore, dairy_free, eggs_free, gluten_free,
    nuts_free, shellfish_free).
ingredient_info(basil, vegan, dairy_free, eggs_free, gluten_free,
    nuts_free, shellfish_free).
```

Listing 4: Example of ingredients info

6 Prolog Rules: Meal Compatibility and Recommendation Logic

The Prolog knowledge base developed for the *Personalized Menu* project includes a set of structured rules that extend beyond basic facts, enabling dynamic reasoning for personalized meal recommendations based on diet preferences, allergen restrictions, and calorie-conscious behavior.

These rules provide a flexible, logic-based framework capable of filtering meals in a way that mirrors realistic guest requirements.

6.1 Allergies Restriction Rules

This rule block defines Prolog predicates that verify whether a given ingredient complies with certain dietary restrictions or allergen exclusions.

Each predicate (is_lactose_free, is_eggs_free, etc.) takes an ingredient as input.

It then checks the corresponding fact in ingredient_info, which stores attributes of each ingredient (like diet type, dairy status, egg status, gluten status, nut status, and shellfish status).

The underscores _ are used as anonymous variables, meaning those fields are ignored in the check.

For example:

is_lactose_free(Ingredient) succeeds if the ingredient_info fact for that ingredient has the dairy_free tag in the correct position.

Similarly, is_gluten_free(Ingredient) succeeds if the ingredient is tagged as gluten_free, and so on.

Rule logic:

```
is_lactose_free(Ingredient) :- ingredient_info(Ingredient, _, dairy_free
   , _, _, _, _).
is_eggs_free(Ingredient) :- ingredient_info(Ingredient, _, _, eggs_free,
    _, _, _).
is_gluten_free(Ingredient) :- ingredient_info(Ingredient, _, _, _,
   gluten_free, _, _).
is_nuts_free(Ingredient) :- ingredient_info(Ingredient, _, _, _, _,
   nuts_free, _).
is_shellfish_free(Ingredient) :- ingredient_info(Ingredient, _, _, _, _,
    _, shellfish_free).
```

Listing 5: Allergies restriction rules

6.2 Diet Compatibility

This block defines dietary compatibility rules for different diet types using the `can_eat` predicate.

Carnivore

`can_eat(carnivore, Ingredient)`: A carnivore diet places no restrictions beyond what is listed in `ingredient_info`.

- Any ingredient is acceptable, regardless of whether it is meat, dairy, or plant-based.

Vegetarian

`can_eat(vegetarian, Ingredient)`: A vegetarian can eat ingredients that are either:

- Explicitly labeled as `vegan`, or labeled as `vegetarian` (e.g., dairy, eggs, or plant-based).

Vegan

`can_eat(vegan, Ingredient)`: A vegan can only eat ingredients explicitly marked as `vegan` in `ingredient_info`.

Example mapping:

```
can_eat(carnivore, Ingredient) :- ingredient_info(Ingredient, _, _, _, _, _, _).
can_eat(vegetarian, Ingredient) :-
    ingredient_info(Ingredient, vegan, _, _, _, _, _);
    ingredient_info(Ingredient, vegetarian, _, _, _, _, _).
can_eat(vegan, Ingredient) :- ingredient_info(Ingredient, vegan, _, _, _, _, _).
```

Listing 6: Diet compatibility rules

6.3 Matching Ingredient

This rule defines how to check whether a single ingredient matches a list of user preferences (dietary choices or allergen restrictions).

```
ingredient_satisfies(Ingredient, Preferences) :-
    (\+ member(lactose_free, Preferences); is_lactose_free(Ingredient)),
    (\+ member(eggs_free, Preferences); is_eggs_free(Ingredient)),
    (\+ member(gluten_free, Preferences); is_gluten_free(Ingredient)),
    (\+ member(nuts_free, Preferences); is_nuts_free(Ingredient)),
    (\+ member(shellfish_free, Preferences); is_shellfish_free(
        Ingredient)),
    (\+ member(vegan, Preferences); can_eat(vegan, Ingredient)),
    (\+ member(vegetarian, Preferences); can_eat(vegetarian, Ingredient)
        ),
    (\+ member(carnivore, Preferences); can_eat(carnivore, Ingredient)).
```

Listing 7: Ingredient satisfies

Explanation:

- member(X, Preferences): checks if a restriction/diet (X) is part of the user's preferences.
- \+ member(X, Preferences): means the preference is not required. In that case, the condition is automatically satisfied.
- If the preference is in the list, then the second part of the condition must hold (e.g., is_lactose_free(Ingredient)).

For example:

- If "lactose_free" is in the preferences → the ingredient must be lactose-free.
- If it's not in the preferences → any ingredient passes this check.

The same applies for:

- eggs_free, gluten_free, nuts_free, shellfish_free → allergen restrictions.
- vegan, vegetarian, carnivore → diet type compatibility, checked with can_eat/2.

6.4 Overall Meal Compatibility

This Prolog rule checks whether an entire meal meets a set of dietary or allergen preferences:

```
meal_satisfies_preferences(Meal, Preferences) :-
    meal(Meal, Ingredients, _),
    forall(member(Ingredient, Ingredients), ingredient_satisfies(
        Ingredient, Preferences)).
```

Listing 8: Meal compatibility

Explanation of the rule `meal_satisfies_preferences`:

1. `meal(Meal, Ingredients, _)`
   - Retrieves the list of `Ingredients` for the meal `Meal`.
   - The underscore `_` indicates that the calorie information is ignored, as this rule only checks dietary preferences.
2. `forall(member(Ingredient, Ingredients), ingredient_satisfies(Ingredient, Preferences))`
   - `member(Ingredient, Ingredients)` iterates over each ingredient in the meal.
   - `ingredient_satisfies(Ingredient, Preferences)` checks whether the ingredient satisfies the given preferences (such as lactose-free, gluten-free, vegan, etc.).
   - `forall` succeeds **only if all ingredients** satisfy the preferences. If even one ingredient does not, the rule fails.

6.5 Check calorie level for a meal

```
    meal_by_calories(Meal, calorie_concious) :-
    meal(Meal, _, calorie_concious).

meal_by_calories(Meal, not_calorie_concious) :-
    meal(Meal, _, calorie_concious);
    meal(Meal, _, not_calorie_concious).
```

Explanation:

- The first rule succeeds if the meal `Meal` is labeled as `calorie_concious`.
- The second rule succeeds if the meal is either `not_calorie_concious` or `calorie_concious`, meaning that a person who is not calorie-conscious can eat any meal.
- The underscore `_` ignores the ingredients for this check.
- The semicolon `;` represents logical OR in Prolog.

6.5.1 Find the List of Meals

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

Explanation:

- **Purpose:** Collect all meals that satisfy a given set of dietary preferences and a calorie level.
- **Arguments:**
  - `Preferences`: list of dietary restrictions or diet types (e.g., `[vegan, lactose_free]`).
  - `CalorieLevel`: either `calorie_concious` or `not_calorie_concious`.
  - `Meals`: the resulting list of meals that satisfy the criteria.
- **Details:**
  - `meal(Meal, _, _)` considers each meal defined by `meal`.
  - `meal_satisfies_preferences(Meal, Preferences)` checks that all ingredients meet the user's preferences.
  - `meal_by_calories(Meal, CalorieLevel)` checks that the meal matches the requested calorie level.
  - `findall` collects all matching `Meal` values into the list `Meals`.
- **Summary:** Returns a list of meals that a person can eat based on their dietary restrictions and calorie preferences.

7 Knowledge Representation Technologies

7.1 Ontology Engineering Role

In the context of knowledge-based systems, **ontology engineering** plays a foundational role by enabling the structured representation of domain-specific knowledge. Rather than relying on informal or ad hoc models, this discipline provides a formal approach to define and organize concepts within a given field.

An **ontology** serves as a clear and machine-interpretable specification of a domain's key elements. It establishes the types of entities involved, the relationships among them, and the logical rules that govern their interactions.

For our project, centered on personalized meal recommendations, ontology engineering allowed us to explicitly model not only **meals and ingredients**, but also **guest profiles, dietary restrictions, allergies**, and **preference patterns**. This structured framework is essential for supporting reasoning tasks, enabling the system to infer which meals are suitable (or not) for each individual.

By capturing both the static structure and dynamic constraints of the domain, the ontology becomes the backbone of intelligent, rule-based personalization.

7.2 Turtle Syntax

To represent the ontology developed in our project, we used the **Turtle (TTL)** serialization format. Turtle is a widely adopted, human-readable syntax for expressing data in **RDF (Resource Description Framework)**, the standard model for structuring information about web resources using **triples**, composed of a subject, predicate, and object.

What makes Turtle particularly suitable for ontology engineering is its ability to define **prefixes** that abbreviate long **URIs (Uniform Resource Identifiers)**. This feature significantly improves both the readability and maintainability of the ontology, especially when compared to more verbose alternatives like RDF/XML. In our project, this feature was not only useful for clarity, but also for keeping the ontology, the SHACL shapes, and the validation reports as separate yet interoperable `.ttl` files, which facilitated querying (SPARQL), reasoning (SWRL), and validation (SHACL).

7.3 Protégé

The ontology was developed using **Protégé**, a widely used open-source tool for ontology modeling developed by **Stanford University**. Protégé provides a robust and user-friendly environment for creating, editing, and managing ontologies. It supports a range of ontology languages, including **OWL (Web Ontology Language)** and **RDF (Resource Description Framework)**. Using Protégé, we were able to formally define domain knowledge through the creation of **classes, object properties, data properties**, and **individuals**. Beyond its general readability, **Protégé** was particularly useful in our project to apply reasoning via **SWRL rules** and to validate **SHACL shapes** through the **SHACL4Protege plugin**. This made Protégé the central environment for ontology development, complemented by external tools such as **pySHACL** for generating formal validation reports.

7.4 Our Ontology

Our ontology is structured around a clear and well-defined class hierarchy, which provides the foundation for consistent and scalable knowledge representation in the project. The hierarchy distinguishes between core concepts such as **Guest**, **Meal**, **Ingredient**, and **IngredientUsage**, while preferences are modeled through the class **GuestFoodPreference**, further specialized into **GuestCaloricPreference** and **GuestFoodTypePreference**. The latter is in turn divided into two relevant branches: **Allergen** and **Diet**, enabling a finer-grained representation of dietary restrictions and intolerances.

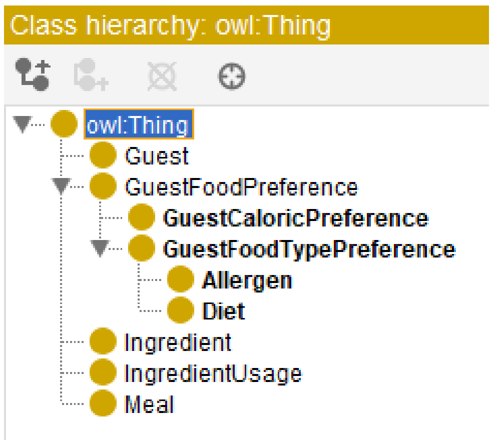


Figure 7: Core class hierarchy defined in Protégé.

This organization allows us to capture both nutritional constraints (e.g., calorie-conscious profiles) and dietary restrictions (e.g., vegetarian, vegan, gluten-free, lactose-free), as well as allergies (e.g., avoidGluten, avoidDairy). In this way, the domain knowledge is represented in a semantically precise and extensible manner. Such a structured approach is crucial not only for maintaining clarity during the modeling phase but also for enabling effective reasoning, inference, and personalized meal recommendations within the system.

7.5 Classes

As previously introduced, the figure highlights the structure of the ontology's class hierarchy. Two core classes can be identified: **Guest** and **Meal**, representing, respectively, the user who interacts with the system and the dish offered by the menu. However, in order to determine which meals are suitable for a specific guest, it is essential to introduce additional parameters that enable user profiling and the identification of dietary preferences or restrictions.

To address this, the ontology includes several supporting classes such as **Ingredient**, **IngredientUsage**, and **GuestFoodPreference**. The latter is further divided into **GuestCaloricPreference** and **GuestFoodTypePreference**, the last mentioned branching into two specific categories: **Allergen** (e.g., avoidGluten, avoidDairy) and **Diet** (e.g., vegetarian, vegan). This structure allows the system to represent both nutritional constraints and dietary restrictions in a fine-grained and extensible manner.

The next step involves a detailed analysis of the main components of the ontology in order to understand their specific role and contribution to the personalized meal recommendation process. In particular, this includes the **classes**, which define the conceptual categories of the domain (e.g., **Meal**, **Guest**, **Ingredient**, **Diet**); the **object properties**, which represent semantic relationships between individuals belonging to different classes (e.g., a guest *GuestHasFoodType*, an ingredient *IngredientNotCompatibleWithFoodTypePreference*, a meal *IngredientUsageInMeal*); and the **data properties**, which associate individuals with literal values such as strings or numbers (e.g., *IngredientHasCalories*, *MealHasCalories*, or *IngredientUsageInGrams*). Finally, the **individuals** are the concrete instances of these classes (e.g., *Claudio* as a Guest, *margherita_pizza* as a Meal, *cheese* as an Ingredient) and represent the actual data over which reasoning and recommendations are performed. This structural foundation enables a systematic exploration of how each element contributes to the ontology's ability to support intelligent, rule-based decision-making.

7.5.1 Meal Class

The `Meal` class defines the dishes available in the system's menu. It is central to the ontology and serves as the primary entity to which dietary constraints and recommendations are applied. Each meal is described by its **ingredients** (linked through dedicated usage instances), its **nutritional content**, and its **compatibility** with various dietary restrictions.

Object properties:

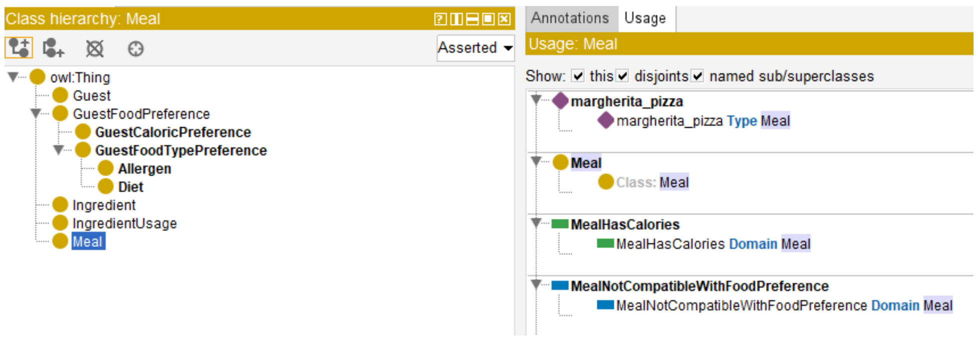
- `IngredientUsageInMeal`: links a meal to the specific `IngredientUsage` instances that describe which ingredients are used and in what quantities.
- `MealNotCompatibleWithFoodPreference`: indicates incompatibility of a meal with certain dietary preferences (e.g., `Gluten_Intolerant`, `Vegan`, `Calorie_Conscious`).

Data properties:

- `MealHasCalories`: associates each instance of the `Meal` class with its caloric value, computed from its ingredients.
- `ObjectHasName`: links a meal to a string literal representing its human-readable name.

Examples of individuals:

- `margherita_pizza`, `pasta_al_pomodoro`, `risotto_ai_funghi`, `baked_sea_bass`, `vegan_pizza`



Class hierarchy: Meal

Asserted

- owl:Thing
  - Guest
  - GuestFoodPreference
    - GuestCaloricPreference
    - GuestFoodTypePreference
      - Allergen
      - Diet
  - Ingredient
  - IngredientUsage
  - Meal

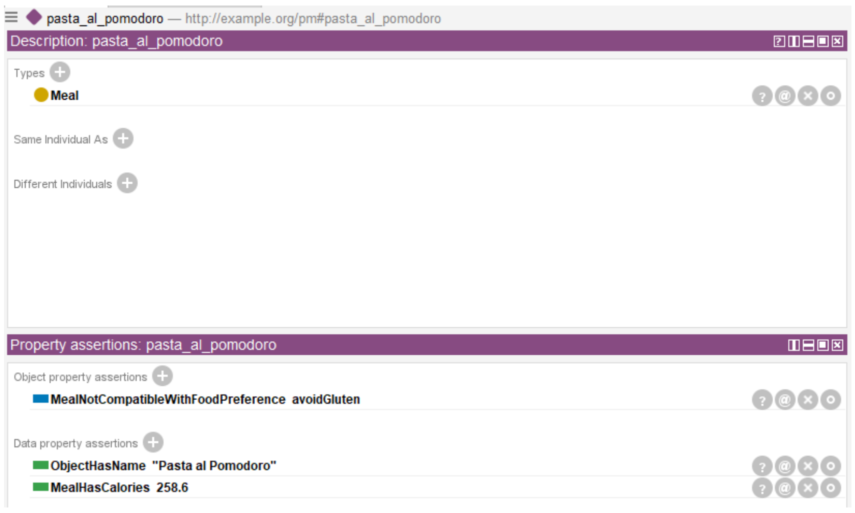
Annotations | Usage

Usage: Meal

Show: ✓ this ✓ disjoints ✓ named sub/superclasses

- margherita_pizza
  - margherita_pizza Type Meal
- Meal
  - Class: Meal
- MealHasCalories
  - MealHasCalories Domain Meal
- MealNotCompatibleWithFoodPreference
  - MealNotCompatibleWithFoodPreference Domain Meal

Figure 8: Meal Class Overview, with Objects Properties, Data Properties and related Individuals



pasta_al_pomodoro — http://example.org/pm#pasta_al_pomodoro

Description: pasta_al_pomodoro

Types
Meal

Same Individual As

Different Individuals

Property assertions: pasta_al_pomodoro

Object property assertions
MealNotCompatibleWithFoodPreference avoidGluten

Data property assertions
ObjectHasName "Pasta al Pomodoro"
MealHasCalories 258.6

Figure 9: Overview of the individual `pasta_al_pomodoro`, with annotated data and object properties.

7.5.2 Guest Class

The **Guest** class represents the users of the system who receive personalized meal recommendations. Each individual belonging to this class is described by properties that capture their dietary preferences and caloric awareness. These characteristics are the foundation for filtering and determining which meals are suitable or incompatible for a given guest.

Object properties:

- `GuestHasFoodType`: links a guest to one or more instances of the `GuestFoodPreference` class, which may represent caloric sensitivity (`GuestCaloricPreference`) or dietary restrictions (`GuestFoodTypePreference`, such as `Vegan`, `Gluten_Intolerant`, `Lactose_Intolerant`).

Data properties:

- `ObjectHasName`: associates each guest with a human-readable string identifying the individual.

Examples of individuals:

- `Claudio`, `Fabio`, `Lorenzo`, `Maria`

7.5.3 Ingredients Class

The `Ingredient` class models the components used in the preparation of meals. It plays a key role in reasoning over dietary restrictions and caloric evaluation, as each ingredient carries both nutritional information and possible incompatibilities with specific food preferences.

Object properties:

- `UsedIngredient`: links an `IngredientUsage` instance to the specific ingredient being used in a meal.
- `IngredientNotCompatibleWithFoodTypePreference`: associates an ingredient with dietary preferences or restrictions for which it is not suitable (e.g., `avoidGluten`, `avoidDairy`, `vegan`).

Data properties:

- `IngredientHasCalories`: stores the caloric value of the ingredient (per 100g).
- `ObjectHasName`: denotes the human-readable name of the ingredient.

Examples of individuals:

- `basil`, `mozzarella`, `almonds`, `clams`, `coffee`

7.5.4 GuestFoodPreference Class

The `GuestFoodPreference` class defines the preferences and restrictions that characterize each guest's profile. It enables the ontology to support reasoning about whether a specific meal aligns with the guest's dietary needs or caloric awareness.

Subclasses:

- `GuestCaloricPreference`: captures whether a guest is `CalorieConscious`.
- `GuestFoodTypePreference`: represents general dietary types and restrictions, and is further divided into:
  - `Allergen`: models specific intolerances such as `avoidDairy`, `avoidEggs`, `avoidGluten`, `avoidNuts`, `avoidShellfish`.

  - `Diet`: models broader dietary regimes such as `Vegan`, `Vegetarian`, `Carnivore`.

Object properties:

- `GuestHasFoodType`: links a guest to one or more of their preferences or restrictions (caloric or dietary).

Examples of individuals:

- `CalorieConscious`, `NotCalorieConscious`, `avoidGluten`, `avoidDairy`, `Vegan`, `Carnivore.`

7.5.5 IngredientUsage Class

The `IngredientUsage` class models the relationship between a `Meal` and the specific `Ingredients` it contains, including the quantity used. This class acts as a linking node that allows the ontology to capture nutritional details more precisely, since each meal is composed of multiple `IngredientUsage` instances that specify which ingredient is included and in what amount.

Object properties:

- `IngredientUsageInMeal`: links the usage instance to the meal where the ingredient is used.
- `UsedIngredient`: specifies which ingredient is being used in the meal.

Data properties:

- `IngredientUsageInGrams`: records the quantity (in grams) of the ingredient used in the meal.

Examples of individuals:

- `usage_margherita_pizza_mozzarella`, `usage_bacon_wrapped_chicken_salt`.

7.6 SWRL Rules

In the context of our project, we integrated **SWRL (Semantic Web Rule Language)** to extend the reasoning capabilities of OWL ontologies. SWRL enables the definition of logical implications by combining class expressions and property relations through rules structured as:

$$\text{antecedent (body)} \rightarrow \text{consequent (head)}$$

The body contains conditions that must be satisfied for the rule to apply, while the head specifies the inferred conclusion. Both sections may consist of multiple predicates combined conjunctively.

We applied SWRL rules to capture implicit knowledge and to support dynamic, rule-based personalization of meal recommendations based on **guest profiles, dietary restrictions, and nutritional thresholds**. Unlike static OWL axioms, these rules activate during reasoning and generate new inferred triples, such as `MealNotCompatibleWithFoodPreference` or calorie-based properties.

Reasoning was performed in **Protégé** using the **HermiT 1.4.3.456** reasoner, which supports the execution of SWRL rules. This setup allowed us to evaluate rules involving numerical thresholds (e.g., total calories of a meal) as well as logical constraints (e.g., excluding meals that contain ingredients incompatible with a guest's food preferences). It is important to note that the results of SWRL reasoning are not persisted in the ontology by default, but are visible during classification. This ensures that recommendations remain flexible and can adapt dynamically to different reasoning tasks.

7.6.1 Implemented Rules

Below we can see a detailed explanation of each SWRL rule implemented in our ontology, covering allergen-based restrictions, caloric preferences, and compatibility with dietary regimes.

7.6.2 1. Gluten Intolerance Check

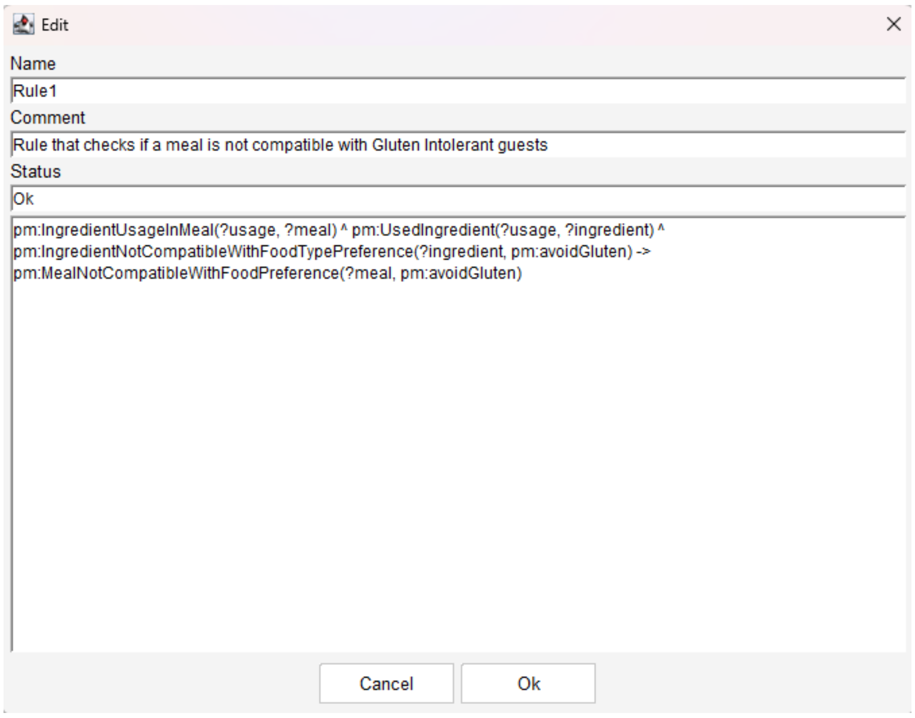
Rule:

```
pm:IngredientUsageInMeal(?usage, ?meal) ^
pm:UsedIngredient(?usage, ?ingredient) ^
pm:IngredientNotCompatibleWithFoodTypePreference(?ingredient, pm:
    avoidGluten)
-> pm:MealNotCompatibleWithFoodPreference(?meal, pm:avoidGluten)
```

Listing 9: Gluten intolerance exclusion rule

Description: If a `Meal` includes an `Ingredient` that is incompatible with the `avoidGluten` preference, then the meal is inferred as not compatible with gluten-intolerant guests.

Use case: If `pasta_alla_bolognese` contains `wheat`, which is incompatible with `avoidGluten`, the meal will be marked as unsuitable for gluten-intolerant guests.



Name
Rule1
Comment
Rule that checks if a meal is not compatible with Gluten Intolerant guests
Status
Ok

```
pm:IngredientUsageInMeal(?usage, ?meal) ^ pm:UsedIngredient(?usage, ?ingredient) ^
pm:IngredientNotCompatibleWithFoodTypePreference(?ingredient, pm:avoidGluten) ->
pm:MealNotCompatibleWithFoodPreference(?meal, pm:avoidGluten)
```

Figure 10: SWRL Rule Overview

7.6.3 2. Lactose Intolerance Check

```
pm:IngredientUsageInMeal(?usage, ?meal) ^
pm:UsedIngredient(?usage, ?ingredient) ^
pm:IngredientNotCompatibleWithFoodTypePreference(?ingredient, pm:
    avoidDairy)
-> pm:MealNotCompatibleWithFoodPreference(?meal, pm:avoidDairy)
```

Listing 10: Lactose intolerance exclusion rule

Description: If a `Meal` includes an `Ingredient` that is incompatible with the `avoidDairy` preference, then the meal is inferred as not compatible with lactose-intolerant guests.

Use case: If `margherita_pizza` contains `mozzarella`, which is incompatible with `avoidDairy`, the meal will be excluded for lactose-intolerant guests.

7.6.4 3. Vegetarian Preference Check

```
pm:IngredientUsageInMeal(?usage, ?meal) ^
pm:UsedIngredient(?usage, ?ingredient) ^
pm:IngredientNotCompatibleWithFoodTypePreference(?ingredient, pm:
    vegetarian)
-> pm:MealNotCompatibleWithFoodPreference(?meal, pm:vegetarian)
```

Listing 11: Vegetarian exclusion rule

Description: If a `Meal` contains an `Ingredient` marked as incompatible with the `vegetarian` preference, the meal is inferred as not suitable for vegetarian guests.

Use case: A meal like `baked_sea_bass`, which includes fish, will be marked as incompatible with the `vegetarian` preference.

7.6.5 4. Vegan Preference Check

```
pm:IngredientUsageInMeal(?usage, ?meal) ^
pm:UsedIngredient(?usage, ?ingredient) ^
pm:IngredientNotCompatibleWithFoodTypePreference(?ingredient, pm:vegan)
-> pm:MealNotCompatibleWithFoodPreference(?meal, pm:vegan)
```

Listing 12: Vegan exclusion rule

Description: If a `Meal` contains any `Ingredient` incompatible with the `vegan` preference, the meal is inferred as not suitable for vegan guests.

Use case: A dish such as `carbonara`, which contains `egg` and `bacon`, will be excluded for guests with the `vegan` preference.

7.7 SHACL Shapes

SHACL (Shapes Constraint Language) is a W3C standard used to validate RDF data by defining structural and logical constraints, known as **shapes**. These shapes describe conditions that RDF instances must satisfy, ensuring consistency and preventing semantic or structural errors within the ontology. In this project, SHACL was used to validate both the structure and semantics of the ontology, checking that each class and property is used consistently according to the defined modeling rules. The shapes were implemented in a dedicated file named `SHACLshapes.ttl`.

7.7.1 1. IngredientShape

Listing:

```
pm:IngredientShape
    a sh:NodeShape ;
    sh:targetClass pm:Ingredient ;
    sh:property [
        sh:path pm:IngredientHasCalories ;
        sh:datatype xsd:integer ;
        sh:minCount 1 ;
        sh:message "Every Ingredient must have at least one calorie
            value (integer)." ;
    ] ;
    sh:property [
        sh:path pm:ObjectHasName ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
        sh:message "Every Ingredient must have a name." ;
    ] .
```

Listing 13: IngredientShape definition

Description: Ensures that every `Ingredient` individual has a valid name and a numeric calorie value. This guarantees data completeness for all ingredients used in meals.

7.7.2 2. MealShape

Listing:

```
pm:MealShape
    a sh:NodeShape ;
    sh:targetClass pm:Meal ;
    sh:property [
        sh:path [ sh:inversePath pm:IngredientUsageInMeal ] ;
        sh:minCount 1 ;
        sh:message "Each Meal must be linked (inverse) to at least one
           IngredientUsage." ;
    ] ;
    sh:property [
        sh:path pm:ObjectHasName ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
        sh:message "Each Meal must have a name." ;
    ] .
```

Listing 14: MealShape definition

Description: Validates that each `Meal` individual is connected to at least one `IngredientUsage` and has a defined name. This ensures that every dish in the ontology is both labeled and compositionally defined.

7.7.3 3. MealCaloriesShape

Listing:

```
pm:MealCaloriesShape
    a sh:NodeShape ;
    sh:targetClass pm:Meal ;
    sh:property [
        sh:path pm:MealHasCalories ;
        sh:datatype xsd:decimal ;
        sh:minInclusive 0 ;
        sh:message "Meal calories must be greater than or equal to 0." ;
    ] .
```

Listing 15: MealCaloriesShape definition

Description: Checks that every `Meal` has a calorie value greater than or equal to zero, ensuring correct numerical data integrity.

7.7.4 4. GuestShape

Listing:

```
pm:GuestShape
    a sh:NodeShape ;
    sh:targetClass pm:Guest ;
    sh:property [
        sh:path pm:GuestHasFoodType ;
        sh:minCount 1 ;
        sh:message "Each Guest must have at least one FoodType." ;
    ] .
```

Listing 16: GuestShape definition

Description: Requires that each `Guest` individual be associated with at least one `FoodType` instance. This ensures that every guest in the system has a defined dietary profile or restriction.

7.7.5 5. FoodPreferenceShape

Listing:

```
pm:FoodPreferenceShape
    a sh:NodeShape ;
    sh:targetClass pm:FoodTypePreference ;
    sh:property [
        sh:path pm:ObjectHasName ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
        sh:message "Each FoodTypePreference must have a name." ;
    ] .
```

Listing 17: FoodPreferenceShape definition

Description: Ensures that all instances of `FoodTypePreference` (including subclasses such as `Diet` and `Allergen`) are properly labeled with a name.

7.7.6 7.6.6 SHACL Validation Process

To ensure the structural and semantic consistency of the ontology, a complete validation workflow was performed using the **SHACL (Shapes Constraint Language)** standard. Two complementary validation approaches were adopted: an **interactive validation** directly within **Protégé** using the **SHACL4Protege Constraint Validator** plugin, and an **external validation** through the **pySHACL** command-line tool for formal and reproducible reporting.

The internal validation in Protégé allowed real-time feedback on shape conformance during ontology modeling. Each shape defined in the `SHACLshapes.ttl` file was individually tested, including `IngredientShape`, `MealShape`, `MealCaloriesShape`, `GuestShape`, and `FoodPreferenceShape`. Initially, several violations were detected (mainly due to naming mismatches such as `GuestHasFoodType` vs. `GuestHasFoodTypePreference`); after refinement of object property alignments, all violations were resolved and the validation concluded with **0 remaining issues**.

To produce a structured validation report, the ontology was subsequently validated externally using the **pySHACL** library. This tool executes SHACL constraints under different entailment regimes, supporting datatype validation and OWL reasoning. The following command was used to perform validation with OWL 2 RL inference and advanced SHACL features enabled:

```
pyshacl -d pm.ttl -s SHACLshapes.ttl -m -i owlrl \
  -o reports/validationReport.json -f json-ld
```

Listing 18: pySHACL validation command

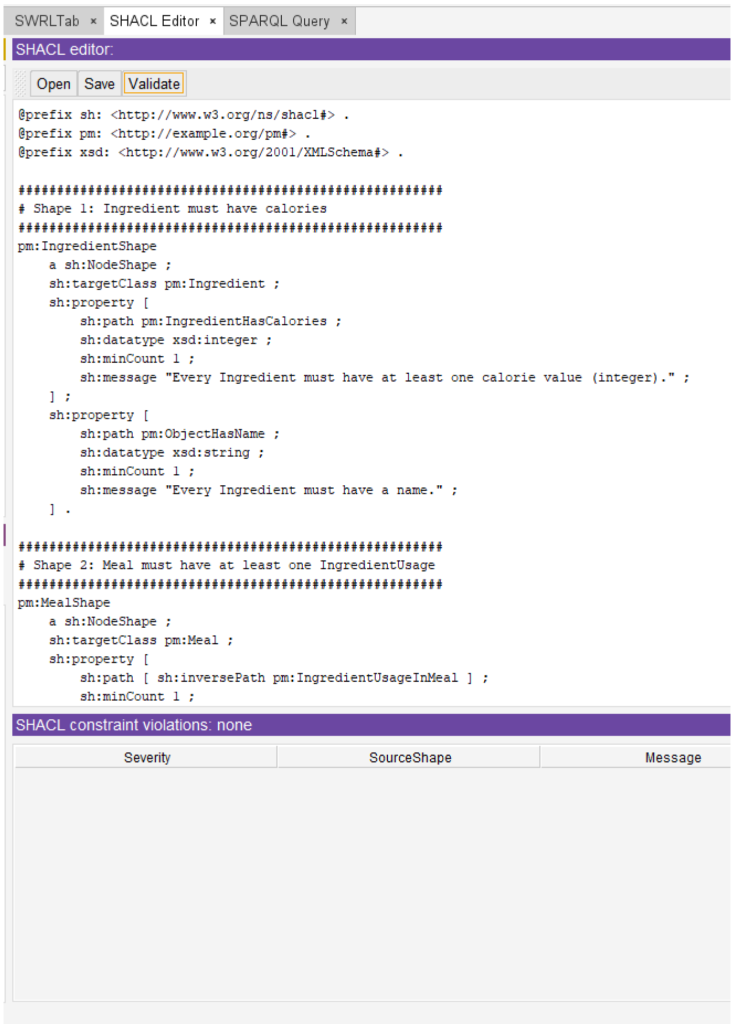
The `-d` parameter specifies the ontology file (data graph), `-s` indicates the SHACL shapes graph, `-m` enables SHACL advanced mode, `-i owlrl` activates OWL 2 RL inferencing, and `-o / -f` export a structured report in **JSON-LD** format. An additional report was generated in **Turtle** format for semantic traceability. The resulting `validationReport.json` conforms to the official SHACL validation report model, containing the overall status (`sh:conforms = true`) and, if applicable, detailed descriptions of constraint violations with their paths, targets, and messages.

The combination of internal (Protégé) and external (pySHACL) validation confirmed the ontology's full compliance with the defined structural constraints. All tested shapes successfully validated that:

- every `Ingredient` has a name and a caloric value;
- every `Meal` is connected to at least one `IngredientUsage`;
- all `MealHasCalories` values are numeric and non-negative;

- each `Guest` is associated with at least one food preference;
- every `FoodTypePreference` has a defined name.

Both validation methods returned a positive result (`Conforms: True`), confirming the structural soundness and reliability of the ontology for subsequent reasoning, SWRL rule evaluation, and SPARQL querying phases.

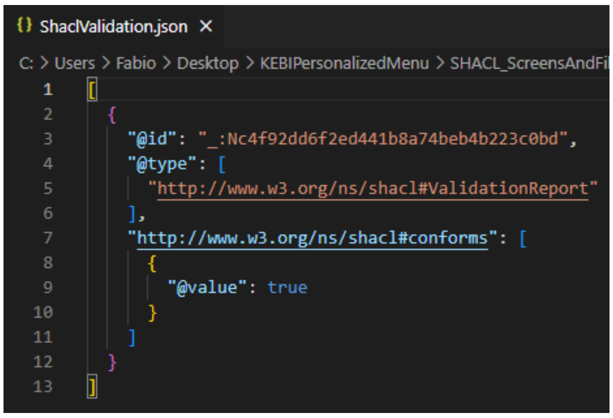


```
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix pm: <http://example.org/pm#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

########################################################
# Shape 1: Ingredient must have calories
########################################################
pm:IngredientShape
    a sh:NodeShape ;
    sh:targetClass pm:Ingredient ;
    sh:property [
        sh:path pm:IngredientHasCalories ;
        sh:datatype xsd:integer ;
        sh:minCount 1 ;
        sh:message "Every Ingredient must have at least one calorie value (integer)." ;
    ] ;
    sh:property [
        sh:path pm:ObjectHasName ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
        sh:message "Every Ingredient must have a name." ;
    ] .

########################################################
# Shape 2: Meal must have at least one IngredientUsage
########################################################
pm:MealShape
    a sh:NodeShape ;
    sh:targetClass pm:Meal ;
    sh:property [
        sh:path [ sh:inversePath pm:IngredientUsageInMeal ] ;
        sh:minCount 1 ;
```

SHACL constraint violations: none

Severity	SourceShape	Message

Figure 11: Validation in Protégé using the SHACL4Protege Constraint Validator, showing 0 violations across all defined shapes.


ShaclValidation.json
C: > Users > Fabio > Desktop > KEBIPersonalizedMenu > SHACL_ScreensAndFi

```json
[
  {
    "@id": "_:Nc4f92dd6f2ed441b8a74beb4b223c0bd",
    "@type": [
      "http://www.w3.org/ns/shacl#ValidationReport"
    ],
    "http://www.w3.org/ns/shacl#conforms": [
      {
        "@value": true
      }
    ]
  }
]
```

Figure 12: External validation with pySHACL confirming full ontology compliance (Conforms: True).

8 7.7 SPARQL Queries

SPARQL (SPARQL Protocol and RDF Query Language) is the W3C standard language used to query and manipulate data represented in RDF graphs. Within our project, SPARQL was employed to extract relevant information from the ontology, verify reasoning results, and simulate dynamic ontology updates such as calorie computation and preference-based filtering. All queries were executed using **Apache Jena Fuseki**, which provided a SPARQL endpoint to interact with the knowledge base through both SELECT and INSERT operations.

The following prefixes were used throughout all queries:

```sparql
PREFIX pm: <http://example.org/pm#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
```

Listing 19: SPARQL Prefixes used in the project

8.1 7.7.1 Query Set Overview

The queries can be divided into two categories:

- **SELECT queries**, used to retrieve and analyze information (e.g., total calories, vegetarian or gluten-free meals).
- **INSERT queries**, used to simulate dynamic ontology enrichment (e.g., computing total calories, adding compatibility flags).

8.1.1 1. Retrieve Ingredients and Calories

This query extracts all ingredient individuals along with their caloric values, enabling a quick overview of the nutritional database.

```sparql
SELECT ?ingredient ?calories
WHERE {
    ?ingredient rdf:type pm:Ingredient ;
                pm:ObjectHasName ?name ;
                pm:IngredientHasCalories ?calories .
}
```

Listing 20: Retrieve Ingredients and Calories

8.1.2 2. Retrieve the Total Calories of a Meal

Computes the total caloric value of each meal by summing the contribution of its ingredients proportionally to their usage quantities.

```
SELECT ?meal (SUM(?calories) AS ?totalCalories)
WHERE {
    ?usage pm:IngredientUsageInMeal ?meal ;
           pm:UsedIngredient ?ingredient ;
           pm:IngredientUsageInGrams ?quantity .
    ?ingredient pm:IngredientHasCalories ?caloriesPer100Grams .
    BIND((?caloriesPer100Grams / 100.0) * ?quantity AS ?calories)
}
GROUP BY ?meal
```

Listing 21: Retrieve the Total Calories of a Meal

8.1.3 3. Retrieve Calorie-Conscious Meals (550 kcal)

Filters the meals suitable for calorie-conscious guests by restricting results to dishes with total caloric values below or equal to 550 kcal.

```
SELECT ?meal (SUM(?calories) AS ?totalCalories)
WHERE {
    ?usage pm:IngredientUsageInMeal ?meal ;
           pm:UsedIngredient ?ingredient ;
           pm:IngredientUsageInGrams ?quantity .
    ?ingredient pm:IngredientHasCalories ?caloriesPer100Grams .
    BIND((?caloriesPer100Grams / 100.0) * ?quantity AS ?calories)
}
GROUP BY ?meal
HAVING (SUM(?calories) <= 550)
```

Listing 22: Retrieve Calorie-Conscious Meals

8.1.4 4–6. Retrieve Vegetarian, Lactose-Free, and Gluten-Free Meals

The following queries exploit the relation `pm:IngredientNotCompatibleWithFoodTypePreference` to identify meals compatible with specific dietary restrictions. The logic is consistent across all cases: meals are retrieved only if they do not contain any ingredient that violates the given preference.

- **Vegetarian Meals:** meals excluding ingredients incompatible with `pm:vegetarian`.
- **Lactose-Free Meals:** meals excluding ingredients incompatible with `pm:avoidDairy`.
- **Gluten-Free Meals:** meals excluding ingredients incompatible with `pm:avoidGluten`.

```
SELECT DISTINCT ?meal
WHERE {
  ?meal rdf:type pm:Meal .
  FILTER NOT EXISTS {
    ?usage pm:IngredientUsageInMeal ?meal ;
           pm:UsedIngredient ?ingredient .
    ?ingredient pm:IngredientNotCompatibleWithFoodTypePreference pm:
       vegetarian .
  }
}
```

Listing 23: Retrieve Vegetarian Meals (example)

8.1.5 7. Retrieve Calorie-Conscious Vegetarian Meals

Combines caloric constraints with dietary restrictions, retrieving meals that are both vegetarian and under 550 kcal.

```
SELECT ?meal (SUM(?calories) AS ?totalCalories)
WHERE {
  ?meal rdf:type pm:Meal .
  ?usage pm:IngredientUsageInMeal ?meal ;
         pm:UsedIngredient ?ingredient ;
         pm:IngredientUsageInGrams ?quantity .
  ?ingredient pm:IngredientHasCalories ?caloriesPer100Grams .
  BIND((?caloriesPer100Grams / 100.0) * ?quantity AS ?calories)
  FILTER NOT EXISTS {
    ?nonVegUsage pm:IngredientUsageInMeal ?meal ;
                 pm:UsedIngredient ?nonVegIngredient .
    ?nonVegIngredient pm:IngredientNotCompatibleWithFoodTypePreference
        pm:vegetarian .
  }
}
GROUP BY ?meal
HAVING (SUM(?calories) <= 550)
```

Listing 24: Retrieve Calorie-Conscious Vegetarian Meals

8.1.6 8. Retrieve Gluten and Lactose-Free Meals

Applies two exclusion filters simultaneously, identifying meals that contain neither gluten nor dairy ingredients.

```
SELECT DISTINCT ?meal
WHERE {
  ?meal rdf:type pm:Meal .
  FILTER NOT EXISTS {
    ?usage pm:IngredientUsageInMeal ?meal ;
           pm:UsedIngredient ?ingredient .
    ?ingredient pm:IngredientNotCompatibleWithFoodTypePreference pm:
        avoidDairy .
  }
  FILTER NOT EXISTS {
    ?usage pm:IngredientUsageInMeal ?meal ;
           pm:UsedIngredient ?ingredient .
    ?ingredient pm:IngredientNotCompatibleWithFoodTypePreference pm:
        avoidGluten .
  }
}
```

Listing 25: Retrieve Gluten and Lactose-Free Meals

8.1.7 9. Retrieve Meals with Food Preferences and Calories

Returns each meal together with its associated incompatibility preference and caloric value, providing a compact summary of dietary restrictions across the dataset.

```
SELECT ?meal ?preference ?calories
WHERE {
    ?meal rdf:type pm:Meal ;
          pm:MealNotCompatibleWithFoodPreference ?preference ;
          pm:MealHasCalories ?calories .
}
```

Listing 26: Retrieve Meals with Food Preferences and Calories

8.2 7.7.2 Update Queries for Future Expansions

From this point onward, we define two **SPARQL Update** queries that are not strictly required by the current ontology, which is already complete, but may be valuable for future extensions. These updates enable the ontology to automatically compute derived information, such as meal calories, or dynamically mark meals as incompatible with certain preferences.

```
INSERT {
    ?meal pm:MealHasCalories ?totalCalories .
}
WHERE {
    SELECT ?meal (SUM(?calories) AS ?totalCalories)
    WHERE {
        ?usage pm:IngredientUsageInMeal ?meal ;
               pm:UsedIngredient ?ingredient ;
               pm:IngredientUsageInGrams ?quantity .
        ?ingredient pm:IngredientHasCalories ?caloriesPer100Grams .
        BIND((?caloriesPer100Grams / 100.0) * ?quantity AS ?calories)
    }
    GROUP BY ?meal
}
```

Listing 27: Insert Total Calories of Meal in the Ontology

```
INSERT {
    ?meal pm:MealNotCompatibleWithFoodPreference pm:Calorie_Conscious .
}
WHERE {
    SELECT ?meal (SUM(?calories) AS ?totalCalories)
    WHERE {
        ?usage pm:IngredientUsageInMeal ?meal ;
               pm:UsedIngredient ?ingredient ;
               pm:IngredientUsageInGrams ?quantity .
        ?ingredient pm:IngredientHasCalories ?caloriesPer100Grams .
        BIND((?caloriesPer100Grams / 100.0) * ?calories AS ?calories)
    }
    GROUP BY ?meal
    HAVING (?totalCalories >= 550)
}
```

Listing 28: Insert Calorie-Conscious Restriction (>= 550 kcal)

This last query dynamically assigns the `Calorie_Conscious` restriction to meals whose total caloric value exceeds 550 kcal, allowing for scalable updates as the ontology expands.

9 Ontology-Aware Process Modeling

9.1 Agile, Ontology-Based Metamodeling (AOAME)

The Agile, Ontology-Based Metamodeling (AOAME) methodology represents a novel and adaptive framework that merges the iterative principles of agile development with the semantic depth of ontology-driven modeling. Its objective is to enhance the flexibility, precision, and long-term sustainability of modeling languages, particularly in dynamic and knowledge-intensive domains such as business processes and intelligent recommendation systems.

AOAME enables the continuous refinement of modeling languages through short development cycles while ensuring formal alignment with the conceptual structures defined in domain ontologies. This dual nature — agile and semantic — guarantees that models evolve consistently with the underlying knowledge base. Moreover, it fosters collaboration between technical modelers and domain experts, allowing models to remain both formally correct and semantically meaningful.

Within the *Personalized Menu* project, AOAME was used to extend the standard **BPMN 2.0** notation with ontology-based semantics derived from the restaurant domain. This extension made it possible to transform the process model into an ontology-aware system capable of reasoning about guest preferences. In particular, the central BPMN element, the task `Create Personalized Guest Menu` ," was semantically enriched according to AOAME principles. Unlike a static parameterization, the task's attributes are now formally connected to specific ontology classes and individuals that represent guest preferences and dietary restrictions. Through these semantic links, each process execution can reference:

- the `GuestFoodTypePreference` class, used to represent dietary choices such as *Vegetarian*, *Vegan*, or *Carnivore*;
- the `GuestCaloricPreference` class, distinguishing between guests who are *CalorieConscious* or *NotCalorieConscious*;
- the `Allergen` subclass of `GuestFoodTypePreference`, capturing allergen avoidance patterns such as *avoidDairy*, *avoidGluten*, or *avoidNuts*.

Each of these parameters corresponds to an ontology property linked to the BPMN task, ensuring that the data provided by the guest during process execution are represented as RDF triples within the knowledge base. These semantic connections are not merely descriptive: they enable reasoning mechanisms and SPARQL queries to automatically infer compatible meals by traversing relations such as `UsedIngredient`, `IngredientUsageInMeal`, and `MealHasCalories`. In this way, the BPMN model and ontology operate as a unified, knowledge-driven system capable of dynamically adapting menu recommendations to each guest profile.

The application of AOAME in this context demonstrates its ability to connect process modeling and knowledge representation seamlessly. By embedding ontology concepts directly into BPMN elements, AOAME ensures that the process remains consistent with the semantic structure of the domain, while still being flexible enough to adapt to evolving user requirements and contextual data.

9.2 Business Process Modeling with BPMN 2.0

9.2.1 What is BPMN 2.0

Business Process Model and Notation (BPMN) 2.0 is an international standard defined by the Object Management Group (OMG) to model and visualize business processes. It provides a comprehensive set of graphical constructs that represent tasks, events, gateways, and message flows, enabling both technical teams and business stakeholders to understand and communicate process logic effectively.

Although BPMN 2.0 offers powerful tools for modeling generic workflows, it was not originally designed to capture domain-specific semantics. For this reason, its expressive power can be enhanced by integrating ontology-based annotations that convey the meaning of process elements. In the *Personalized Menu* project, AOAME was used to extend BPMN 2.0 with semantic attributes representing key domain concepts such as guest profiles, preferences, and dietary constraints.

9.2.2 Our BPMN Model

The BPMN model developed for this project represents the complete interaction between a guest and the restaurant system, structured across two main lanes: the **Guest Lane** and the **Restaurant System Lane**.

The process begins when the guest scans a QR code provided at the table, triggering the initialization of a personalized interaction. Through the user interface, the guest specifies their food-related preferences, including dietary style, allergen sensitivities, and calorie awareness. These inputs are then encapsulated into semantic parameters and transmitted to the system in the message event labeled `Send Preferences to the System`.

On the system side, the process continues with the task `Receive Preferences`, which records the input data and integrates it into the ontology layer. The core BPMN element — **`Create Personalized Menu for the Guest`** — has been semantically extended following AOAME principles to directly connect with the ontology's conceptual structure.

The task does not store primitive values (e.g., simple strings such as "vegetarian" or "lactose intolerant"), but rather links to formal ontology classes and individuals representing:

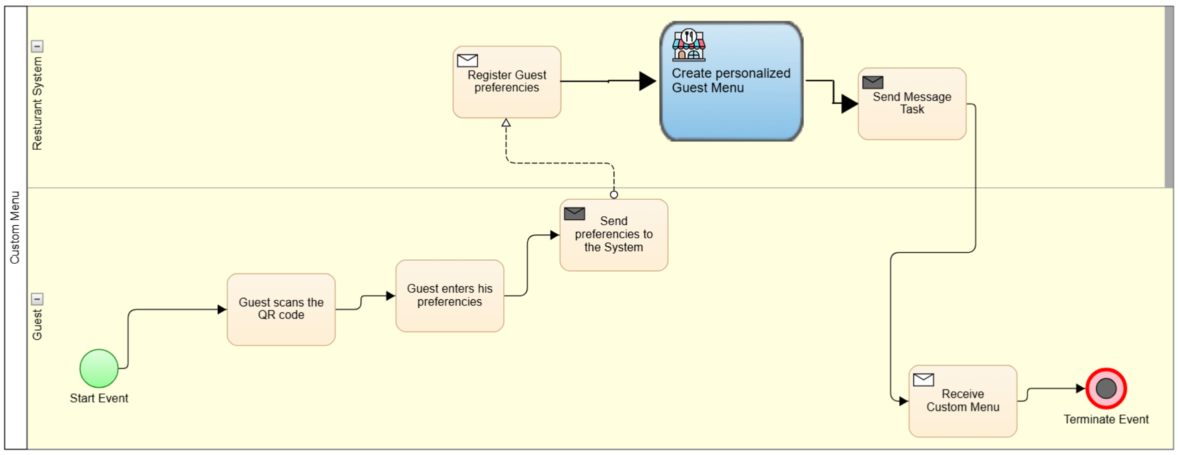
- **GuestFoodTypePreference**, which defines the guest's dietary category (e.g., *Vegetarian*, *Vegan*, *Carnivore*);
- **GuestCaloricPreference**, which specifies whether the guest is *CalorieConscious* or *NotCalorieConscious*;
- **Allergen**, a subclass of `GuestFoodTypePreference`, which represents allergen-related restrictions such as *avoidDairy*, *avoidGluten*, or *avoidNuts*.

These semantic attributes are linked to the process task through the AOAME extension mechanism and serialized as RDF triples in the knowledge base. During execution, they act as dynamic parameters that guide the reasoning process. Through the integration with Jena Fuseki, these parameters are injected into a parameterized SPARQL query that retrieves only the meals compatible with the guest's declared preferences.

This setup allows the BPMN process to interact directly with the ontology, enabling a seamless data flow between the guest input, the reasoning layer, and the final menu generation. The resulting model is not only executable but also semantically interpretable: each decision or query operation is grounded in the ontology's domain logic, ensuring consistency and adaptability to different guest profiles.

Once the preferences are captured, the system uses the semantic connection to query the knowledge graph via **Jena Fuseki**. The SPARQL query dynamically retrieves only the meals compatible with the declared preferences. For instance, a vegetarian guest who is also calorie-conscious would receive a list of light, non-meat dishes such as *margherita pizza* or *grilled vegetables.* This dynamic filtering illustrates how AOAME enables a direct bridge between business process execution and ontology-based reasoning.

The process concludes when the system sends the final personalized menu to the guest, completing a fully semantic, ontology-driven workflow. As illustrated in Figure 13, the model achieves a consistent alignment between the procedural structure of BPMN and the declarative semantics of the underlying ontology, demonstrating AOAME's effectiveness in unifying process modeling and knowledge representation.



Custom Menu
Resturant System
Guest
Register Guest preferencies
Create personalized Guest Menu
Send Message Task
Send preferencies to the System
Guest scans the QR code
Guest enters his preferencies
Start Event
Receive Custom Menu
Terminate Event

Figure 13: BPMN model using AOAME for guest preference integration

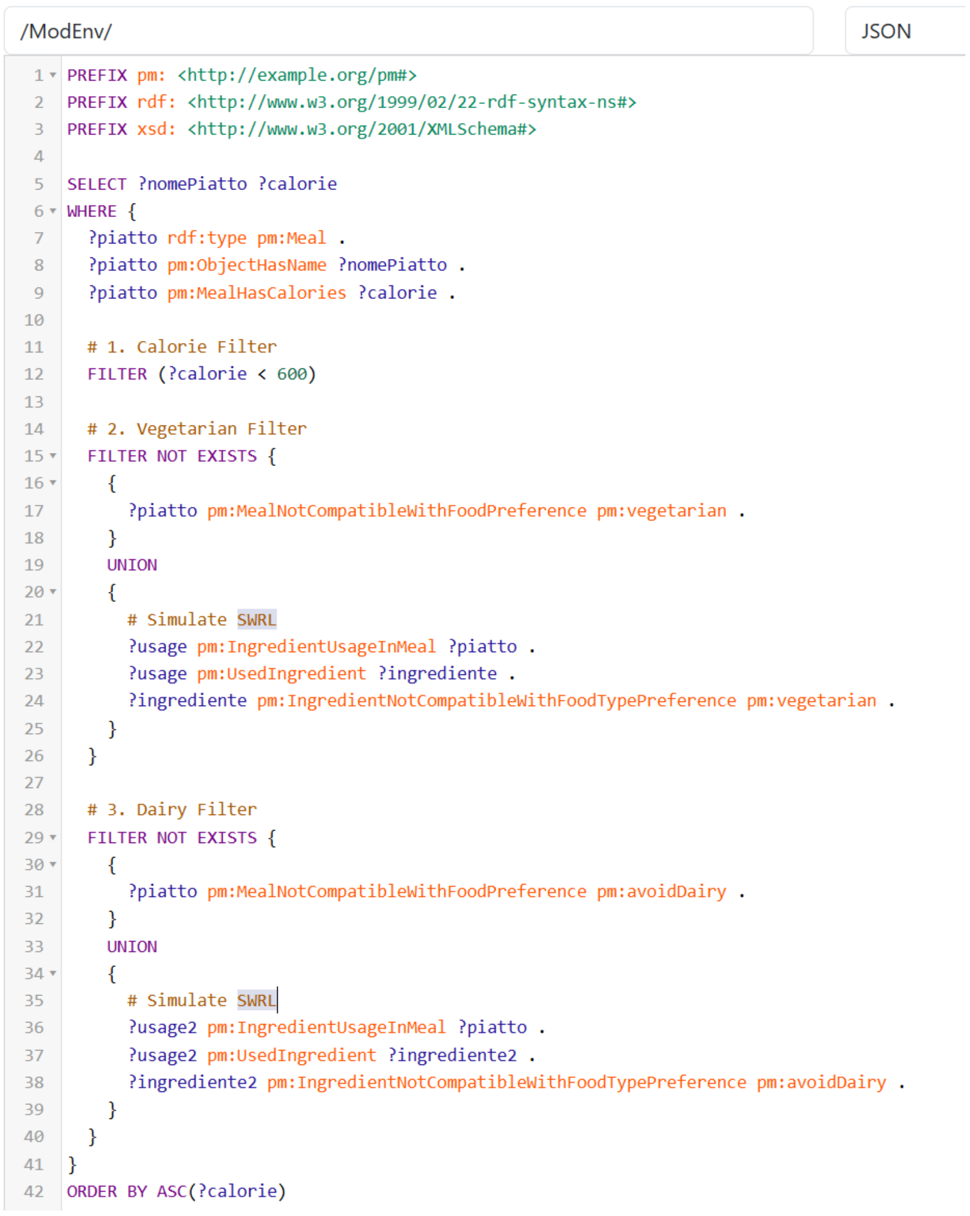
9.3 Jena Fuseki

Jena Fuseki is an open-source SPARQL server and RDF triple store, part of the Apache Jena framework. It is widely used for storing, querying, and managing semantic web data represented as RDF triples (subject–predicate–object). Fuseki provides a SPARQL endpoint that allows applications to query knowledge graphs and retrieve semantically enriched results over HTTP.

In this project, Fuseki acts as the communication interface between the BPMN process and the ontology. Through the semantic annotations defined using AOAME, Fuseki receives parameters such as `DietType`, `CaloriePreference`, and `Allergen` from the process model and injects them into SPARQL queries at runtime. These queries retrieve the most suitable meals for the guest based on ontology-based reasoning, ensuring that the generated recommendations respect dietary rules, allergies, and calorie preferences.

9.3.1 Query in Jena Fuseki

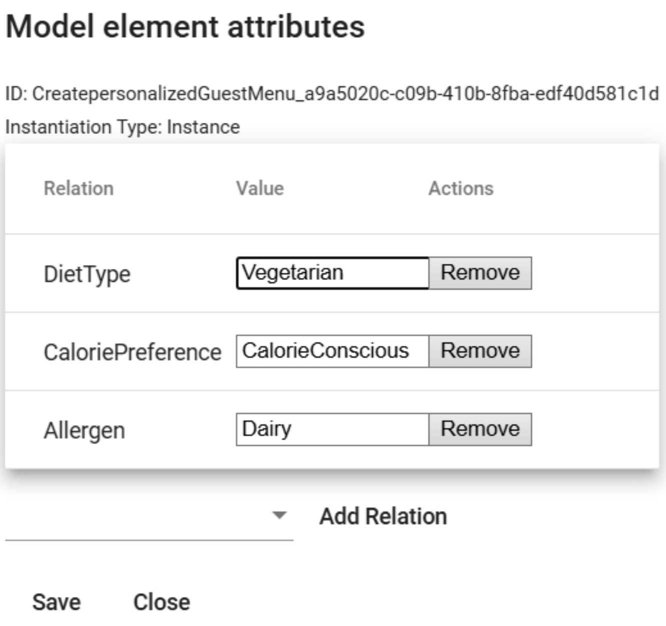
On the **Jena Fuseki** server, a parameterized SPARQL query was implemented to dynamically retrieve a personalized menu from the ontology. This ontology, designed according to the principles of AOAME, includes the classes and properties necessary to manage guest profiles directly within the BPMN process. By linking process attributes (`DietType`, `CaloriePreference`, `Allergen`) to ontology properties, Fuseki can reason over user data and generate tailored menu recommendations.



```
PREFIX pm: <http://example.org/pm#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?nomePiatto ?calorie
WHERE {
  ?piatto rdf:type pm:Meal .
  ?piatto pm:ObjectHasName ?nomePiatto .
  ?piatto pm:MealHasCalories ?calorie .

  # 1. Calorie Filter
  FILTER (?calorie < 600)

  # 2. Vegetarian Filter
  FILTER NOT EXISTS {
    {
      ?piatto pm:MealNotCompatibleWithFoodPreference pm:vegetarian .
    }
    UNION
    {
      # Simulate SWRL
      ?usage pm:IngredientUsageInMeal ?piatto .
      ?usage pm:UsedIngredient ?ingrediente .
      ?ingrediente pm:IngredientNotCompatibleWithFoodTypePreference pm:vegetarian .
    }
  }

  # 3. Dairy Filter
  FILTER NOT EXISTS {
    {
      ?piatto pm:MealNotCompatibleWithFoodPreference pm:avoidDairy .
    }
    UNION
    {
      # Simulate SWRL
      ?usage2 pm:IngredientUsageInMeal ?piatto .
      ?usage2 pm:UsedIngredient ?ingrediente2 .
      ?ingrediente2 pm:IngredientNotCompatibleWithFoodTypePreference pm:avoidDairy .
    }
  }
}
ORDER BY ASC(?calorie)
```

Figure 14: Jena Fuseki query

The query relies on the ontology's extended structure, which acts as a semantic interface between the BPMN model and the knowledge graph. As shown in Figure **??**, the parameters are not hardcoded within the query but are injected at runtime based on the data provided by the BPMN process. This design ensures that the query remains modular and reusable, while the data-driven parameters enable context-sensitive reasoning. For instance, if a guest declares a vegetarian diet and indicates calorie-consciousness, the SPARQL query automatically filters out meals containing meat or high-calorie ingredients, returning options such as *margherita pizza* or *grilled vegetables*.



Model element attributes

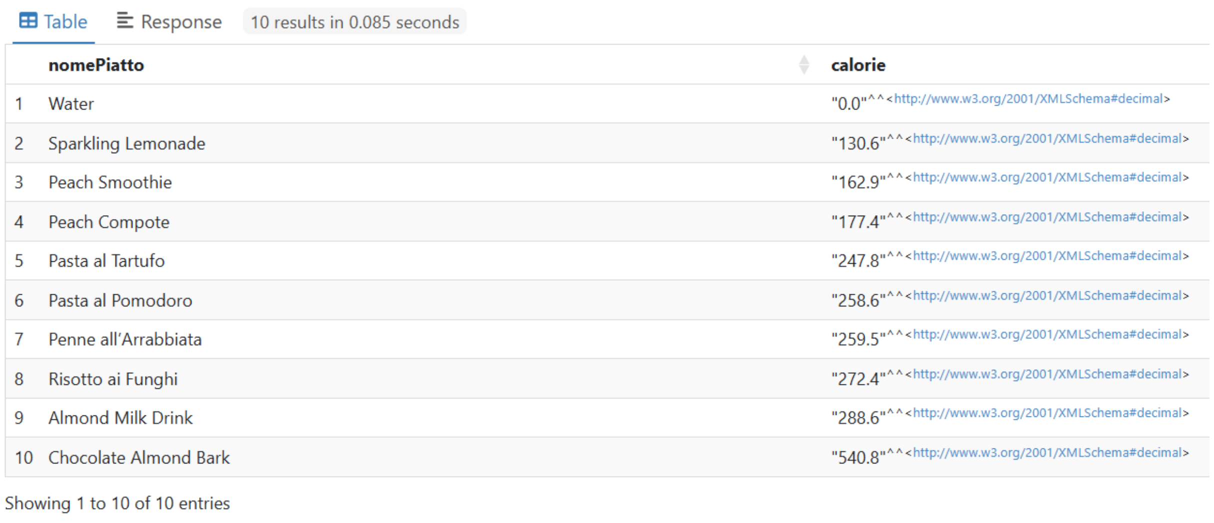
ID: CreatepersonalizedGuestMenu_a9a5020c-c09b-410b-8fba-edf40d581c1d

Instantiation Type: Instance

Relation	Value	Actions
DietType	Vegetarian	Remove
CaloriePreference	CalorieConscious	Remove
Allergen	Dairy	Remove

Figure 15: Semantic annotation of the task `Create personalized Guest Menu` using AOAME. Parameters like `DietType`, `CaloriePreference`, and `Allergen` are attached to enable semantic reasoning.

This approach ensures a seamless integration between process execution and semantic reasoning. As the process runs, the annotated BPMN task transfers guest preferences to the ontology, where the Fuseki engine executes the SPARQL query and returns the corresponding meal recommendations. This real-time exchange confirms the interoperability achieved through AOAME's metamodeling layer.



10 results in 0.085 seconds

	nomePiatto	calorie
1	Water	"0.0"^^<http://www.w3.org/2001/XMLSchema#decimal>
2	Sparkling Lemonade	"130.6"^^<http://www.w3.org/2001/XMLSchema#decimal>
3	Peach Smoothie	"162.9"^^<http://www.w3.org/2001/XMLSchema#decimal>
4	Peach Compote	"177.4"^^<http://www.w3.org/2001/XMLSchema#decimal>
5	Pasta al Tartufo	"247.8"^^<http://www.w3.org/2001/XMLSchema#decimal>
6	Pasta al Pomodoro	"258.6"^^<http://www.w3.org/2001/XMLSchema#decimal>
7	Penne all'Arrabbiata	"259.5"^^<http://www.w3.org/2001/XMLSchema#decimal>
8	Risotto ai Funghi	"272.4"^^<http://www.w3.org/2001/XMLSchema#decimal>
9	Almond Milk Drink	"288.6"^^<http://www.w3.org/2001/XMLSchema#decimal>
10	Chocolate Almond Bark	"540.8"^^<http://www.w3.org/2001/XMLSchema#decimal>

Showing 1 to 10 of 10 entries

Figure 16: Jena Fuseki results

The results displayed in Figure 16 validate the effectiveness of the approach: the system successfully produces a personalized, ontology-driven menu aligned with the guest profile defined within the BPMN process. This confirms that the AOAME-based integration provides a robust foundation for dynamic, knowledge-based recommendation systems.

10 Conclusion - Claudio

The *Personalized Menu* project provided an integrated environment to explore and compare multiple knowledge representation paradigms (Decision Tables, Prolog, and Ontology) demonstrating how each contributes to building an intelligent, adaptive recommendation system. While my main focus was on the process modeling and AOAME integration, this work also allowed me to evaluate how different languages perform when applied to the same domain task: reasoning about personalized food recommendations.

10.1 Integrating Knowledge Representation with Process Modeling

A major achievement of the project was the semantic extension of **BPMN 2.0** through the **Agile Ontology-Based Metamodeling (AOAME)** methodology. This approach enabled the creation of ontology-aware BPMN tasks that directly reference semantic entities such as `DietType`, `Allergen`, and `CaloriePreference`. The result was a process model capable of interacting dynamically with the ontology: guest preferences entered during process execution were automatically passed to **Jena Fuseki**, which in turn executed SPARQL queries to retrieve suitable meals. This integration showed that ontology-based reasoning can be effectively operationalized within process modeling environments, enhancing the adaptability and semantic accuracy of workflows. AOAME thus served as a bridge between declarative knowledge representation and executable business logic, demonstrating the real-world applicability of semantic technologies within BPMN.

10.2 Comparing the Knowledge Representation Approaches

The project offered an opportunity to test three distinct paradigms of knowledge representation and understand their trade-offs in expressiveness, maintainability, and scalability.

Decision Tables (Trisotech) provided a simple and highly interpretable framework for representing decision logic. Their visual nature made them ideal for capturing straightforward business rules, especially those that can be represented as binary or tabular conditions. However, their expressiveness was limited when dealing with multi-dimensional reasoning—such as combining allergens, calorie limits, and dietary restrictions simultaneously. Maintaining consistency across multiple interconnected tables also became cumbersome as the decision logic grew in complexity.

Prolog Logic Programming offered much greater flexibility through its declarative structure, allowing the creation of sophisticated reasoning patterns with compact, recursive rules. It was highly effective for verifying compatibility between meals and guest preferences and for modeling conditional dependencies. Nevertheless, Prolog requires a strong understanding of formal logic, and its integration with other systems (like BPMN or databases) is less direct. It is powerful for inference but less practical for communication with process-oriented tools.

Finally, the **Ontology-based solution (Protégé + Jena Fuseki)** represented the most semantically expressive and interoperable approach. By modeling domain concepts explicitly (Guests, Meals, Ingredients, Preferences) and linking them through object properties, the ontology achieved both clarity and extensibility. The reasoning capabilities provided by **SWRL rules** and **SHACL validation** ensured that new knowledge could be inferred automatically and that data integrity was maintained. Unlike the other two paradigms, the ontology supported seamless integration with the BPMN process via AOAME and SPARQL, enabling real-time, knowledge-driven recommendations. Its main limitation lies in its technical complexity and the need for specialized semantic tools.

10.3 Justifying the Preferred Approach

From a comparative perspective, the ontology-based approach emerged as the most balanced and forward-looking solution. Decision Tables excelled in transparency and communication, Prolog in formal inference, but the ontology unified both interpretability and reasoning within a single semantic framework. It allowed for interoperability across systems, traceability of decisions, and future scalability, all of which are essential for long-term knowledge-based applications. In the specific context of this project, where BPMN, reasoning, and personalization had to coexist, the ontology provided the only architecture capable of connecting all layers consistently, from guest input to real-time menu generation.

10.4 Reflection

This project has been an invaluable opportunity to connect process modeling, logic-based reasoning, and semantic technologies into one coherent system. Working with multiple paradigms not only deepened my understanding of their theoretical underpinnings but also clarified how they complement each other when applied collaboratively. Ultimately, I consider the ontology-based metamodeling approach, combined with AOAME, to be the most sustainable and extensible solution. It ensures that process models are not static diagrams but dynamic, knowledge-aware entities—capable of evolving alongside real-world needs and maintaining semantic integrity across the entire decision-making workflow.

11 Conclusion - Fabio

This project represented a unique opportunity to work across the full spectrum of knowledge-based representation techniques (Decision Tables, Prolog logic programming, and Ontology engineering) allowing a deeper understanding of their respective advantages, limitations, and ideal use cases. While my main contribution focused on ontology design and semantic reasoning, I was actively involved in the integration phase, ensuring that the ontology could interoperate seamlessly with BPMN models through the AOAME methodology and the Jena Fuseki environment.

11.1 Comparative Analysis of Knowledge Representation Approaches

Throughout development, each language demonstrated distinct strengths in the context of personalized menu recommendation. **Decision Tables** (Trisotech) offered excellent readability and accessibility, making it the most intuitive medium for expressing business logic and decisions. Their tabular format made it simple to trace the reasoning behind a recommendation, which is crucial for explainability. However, it lacked the flexibility to handle nested conditions or relational reasoning, which limited its expressiveness as the complexity of guest profiles increased.

Prolog, on the other hand, provided an expressive and logic-driven environment that could model complex dependencies through recursion and rule chaining. Its declarative nature made it ideal for defining formal reasoning patterns such as allergen exclusion and calorie-based filtering. Yet, the procedural burden of maintaining consistency among facts, as well as the lack of an intuitive interface, made Prolog less scalable for continuous system evolution or integration with external tools.

Finally, the **ontology-based approach**—implemented in Protégé and deployed through Jena Fuseki, emerged as the most robust and extensible framework. It combined the clarity of a conceptual model with the inferential power of reasoning. By defining relationships among classes such as `Guest`, `Meal`, and `Ingredient`, and applying **SWRL rules** and **SHACL validation**, the ontology achieved both semantic depth and logical precision. Unlike the other two paradigms, it supported automated inference and interoperability with BPMN via SPARQL queries, enabling a dynamic, knowledge-driven personalization process.

11.2 From Ontology Design to Process Integration

My core responsibility involved designing the ontology and connecting it to the business process model. This included defining the class hierarchy, object properties, and rules governing meal compatibility, but also ensuring that the semantic layer could be operationalized within a real workflow. Working with Claudio, we extended BPMN 2.0 elements through **Agile Ontology-Based Metamodeling (AOAME)**, creating ontology-aware tasks enriched with semantic attributes such as `DietType`, `Allergen`, and `CaloriePreference`. These annotations allowed the BPMN process to communicate directly with the ontology through Jena Fuseki, dynamically adapting recommendations based on the guest profile captured at runtime. This integration proved that ontology-driven reasoning can effectively complement and enrich traditional process modeling by making business workflows semantically aware.

11.3 Justification of the Preferred Approach

After analyzing all three paradigms, I believe that the ontology-based approach provides the most balanced trade-off between formal precision, extensibility, and system interoperability. Decision Tables were highly interpretable but limited in expressive power; Prolog was powerful but isolated and harder to maintain; the ontology, by contrast, unified structure, semantics, and reasoning into a single coherent model, ensuring that knowledge remains both machine-readable and explainable to humans. Moreover, its compatibility with semantic standards (RDF, OWL, SHACL, SWRL, SPARQL) and its seamless integration with BPMN models demonstrated its potential as a foundation for scalable, interoperable, and future-proof personalization systems.

11.4 Final Reflection

This project significantly enhanced my expertise in semantic technologies, collaborative modeling, and cross-paradigm integration. Beyond consolidating my knowledge of ontology design, it reinforced the importance of viewing knowledge representation not as an isolated discipline but as a continuum—from declarative business logic to formal reasoning and semantic process execution. Ultimately, this experience confirmed that ontology-based modeling, when combined with agile metamodeling principles, offers a powerful and sustainable framework for building intelligent, explainable systems capable of real-world personalization and reasoning.

12 Final Considerations

The *Personalized Menu* project has shown that combining knowledge-based systems, logic programming, and semantic process modeling can produce an intelligent and adaptive framework for real-world personalization tasks. By integrating **Decision Tables**, **Prolog logic**, and an **Ontology-based knowledge graph**, further enhanced through **BPMN 2.0** extensions using **AOAME**—the project successfully bridged the gap between structured decision logic, formal reasoning, and semantic interoperability.

Each paradigm played a crucial and complementary role. **Decision modeling** contributed transparency and business interpretability, allowing decision logic to be expressed in a format accessible even to non-technical users. **Prolog** provided formal inference capabilities and recursive reasoning over complex rule structures, making it ideal for modeling logical dependencies such as dietary restrictions or allergen exclusions. Meanwhile, the **ontology and knowledge graph** established a shared, machine-interpretable representation of the domain, enabling automated reasoning, data validation through SHACL, and flexible SPARQL querying. The integration of these layers through **Jena Fuseki** demonstrated the practical viability of connecting reasoning mechanisms with process execution, achieving dynamic, real-time menu recommendations fully aligned with guest profiles.

However, the project also exposed key challenges typical of multi-paradigm systems. Integrating diverse technologies required a high degree of technical coordination and a deep understanding of both logical and semantic principles. Tool interoperability (especially when extending BPMN with AOAME) proved conceptually rich but technically demanding. These challenges underline the importance of developing improved toolchains and standards for connecting business process models with semantic reasoning systems.

Despite these complexities, the system achieved its goal: producing consistent, explainable, and context-aware recommendations. The results validated the scalability of ontology-based metamodeling and demonstrated that intelligent decision support can be made both transparent and adaptive when multiple knowledge paradigms cooperate within a unified architecture.

From a broader perspective, this work reinforces the idea that semantic and process-aware design can transcend the restaurant domain. The same principles could be applied to healthcare, tourism, logistics, or e-commerce; any domain where decisions must adapt dynamically to user profiles and contextual information. Future developments could focus on integrating **machine learning** for predictive preference modeling, automating knowledge graph updates, or extending the ontology to include cultural and nutritional dimensions.

Ultimately, the project provides not only a functioning prototype but also a methodological blueprint for building **intelligent, explainable, and interoperable knowledge-based systems**—where reasoning, process modeling, and semantic understanding converge to deliver personalized, human-centered digital experiences.