University of Camerino

SCHOOL OF SCIENCES AND TECHNOLOGY

Master degree in Computer Science (Class LM-18)

A Knowledge-Based Personalized Menu Recommender

KEBI - Project

Group Member
Diego Marinangeli

Matricola 132676

Supervisors
Prof. Dr. Emanuele Laurenzi
Prof. Dr. Knut Hinklemann

A.A. 2024/2025

Contents

1. Introduction

Digitalization has become a dominant trend in the restaurant sector, with many establishments offering digitized menus accessible through QR codes. While this approach is convenient, it presents challenges for users: smartphone screens are relatively small, and the menus can become difficult to navigate, especially when the number of dishes is large. Furthermore, not all customers are interested in every dish, due to personal dietary preferences or restrictions such as vegetarianism, veganism, allergies, or specific nutritional goals.

The aim of this project is to design a knowledge-based system that filters and recommends dishes according to guest preferences, thereby improving both the dining experience and operational efficiency. By leveraging formalized knowledge representation and reasoning mechanisms, the system is able to provide tailored suggestions that go beyond simple menu listings.

The knowledge representation is modeled using three complementary approaches:

1. **Decision Tables and Decision Requirement Diagrams (DRD)** according to the DMN standard, enabling clear, rule-based decision logic that can be easily understood and maintained by restaurant staff;

2. **Prolog**, using facts and inference rules, allowing for flexible reasoning and the handling of complex dietary constraints through logical deduction;

3. **Knowledge Graph and Ontology**, extended with SWRL rules, SPARQL queries, and SHACL constraints, providing semantic richness, interoperability, and advanced querying capabilities.

Additionally, a meta-modeling approach is applied to BPMN 2.0, adapting process modeling to incorporate decision logic for restaurant menu recommendation. This integration allows the recommendation logic to be directly embedded into operational workflows, ensuring consistency between business processes and intelligent menu suggestions.

One of the key motivations behind this project is to explore how knowledge-based systems can bridge the gap between human expertise and automated decision-making. In the restaurant context, this means capturing culinary knowledge, nutritional information, and customer preferences in a structured and machine-readable format. By doing so, the system can provide not only accurate recommendations but also explainable and traceable reasoning, which is essential for gaining trust from both managers and guests.

Moreover, this project evaluates each approach in terms of practical deployment considerations. Decision tables are highly maintainable and easy to update, making them suitable for rapidly changing menus. Prolog offers expressive reasoning capabilities, especially for constraints that involve complex logical relationships. Ontologies and knowledge graphs provide the highest degree of semantic interoperability and allow the integration of external data sources, such as seasonal ingredient availability or health guidelines.

Finally, the project includes a comprehensive analysis of the advantages and disadvantages of each approach, focusing on aspects such as maintainability, scalability, reasoning power, explainability, and integration with existing systems. By comparing these methodologies, the project highlights best practices and offers insights for future implementations of intelligent recommendation systems in the hospitality industry.

2. Knowledge-Based Solution

This section presents in detail the knowledge-based solution implemented for the personalized menu. As outlined in **Task 1**, the initial approach relied on Decision Tables to filter the restaurant's dishes based on guests' preferences. After defining the Decision Tables, a Prolog-based implementation was provided to create the recommendation system for the personalized menu. Additionally, an ontology was created using Protegé, powered by SPARQL queries and SHACL validation, ensuring data integrity and consistency.

2.1 Decision Tables

The first phase of the solution is built upon a *Knowledge Source* referred to as the *Menu Database*. This source feeds the initial decision, named *Menu*, within which the complete list of restaurant dishes is defined.

Once this central database is established, the system is designed to refine the menu further through specialized decision modules that focus on three main aspects: filtering dishes based on allergenic content, tailoring the selection to match the guest's dietary profile, and aligning dishes with caloric requirements. The module handling allergenic content takes into account common allergens such as lactose, eggs, fish, soy, nuts and gluten. These allergens have been identified as the most significant due to their frequent occurrence and potential to affect a large portion of the clientele.

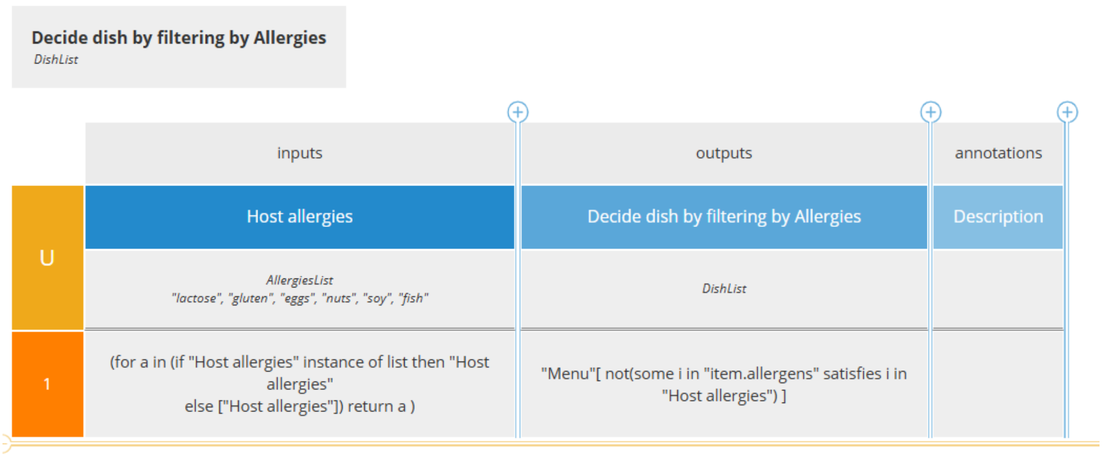
In parallel, the solution considers the guest's dietary profile, accommodating preferences ranging from carnivorous diets to vegetarian, vegan, and pescatarian diets, thereby covering a broad spectrum of culinary tastes. Furthermore, the system evaluates the caloric aspect by categorizing dishes based on their energy content, defining ranges such as *Light* (100-300 kcal), *Balanced* (300-500 kcal), *Rich* (500-1000 kcal), and *High Calories* (above 1000 kcal). This structured approach ensures a wide array of input data for the decision-making process, thereby enhancing the precision of personalized recommendations.

2.1.1 Decision Flow for Menu Customization

After processing the input, the system proceeds through several decision steps to filter the menu:

Filtering by Allergies

The first decision point is Decide Dish by Filtering by Allergies. This decision ensures that the dishes containing ingredients that trigger allergies are excluded from the final



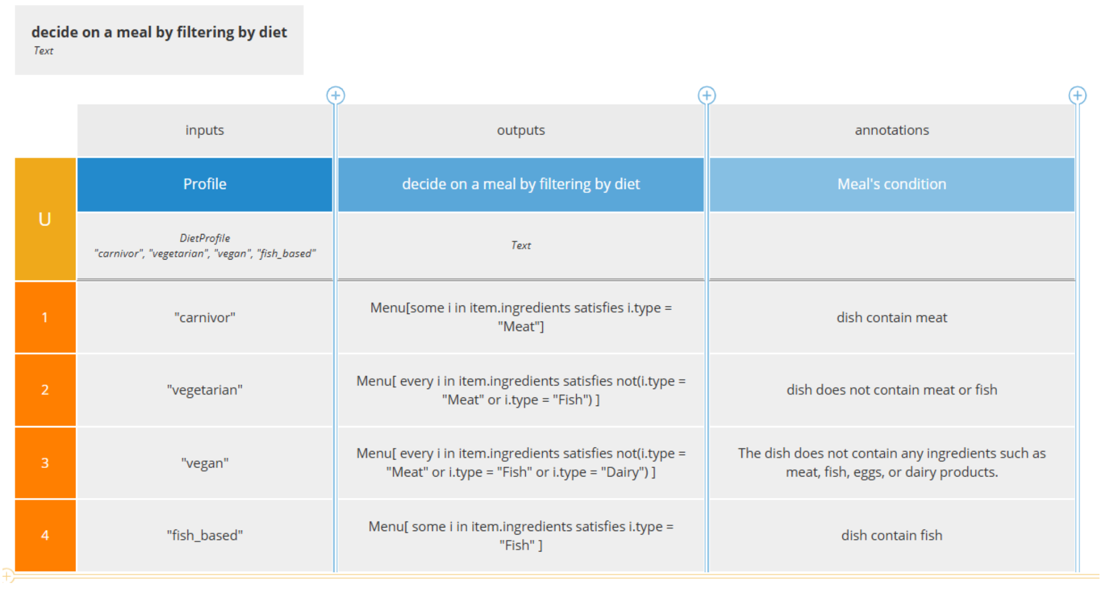
Decide dish by filtering by Allergies
DishList

	inputs	outputs	annotations
U	Host allergies	Decide dish by filtering by Allergies	Description
	AllergiesList *"lactose", "gluten", "eggs", "nuts", "soy", "fish"*	*DishList*	
1	(for a in (if "Host allergies" instance of list then "Host allergies" else ["Host allergies"]) return a)	"Menu"[not(some i in "item.allergens" satisfies i in "Host allergies")]	

selection. For example, if the guest has a dairy allergen, dishes containing ingredients like butter or cheese will be removed from the available options.

Filtering by Dietary Preferences

Next, the system applies the Decide on a Meal by Filtering by Diet decision. This decision filters out dishes that do not match the guest's declared dietary profile, such as vegetarian, pescatarian, or gluten-free diets. For example, if the guest is vegetarian, dishes like "Lasagna" which contain meat, will be excluded.

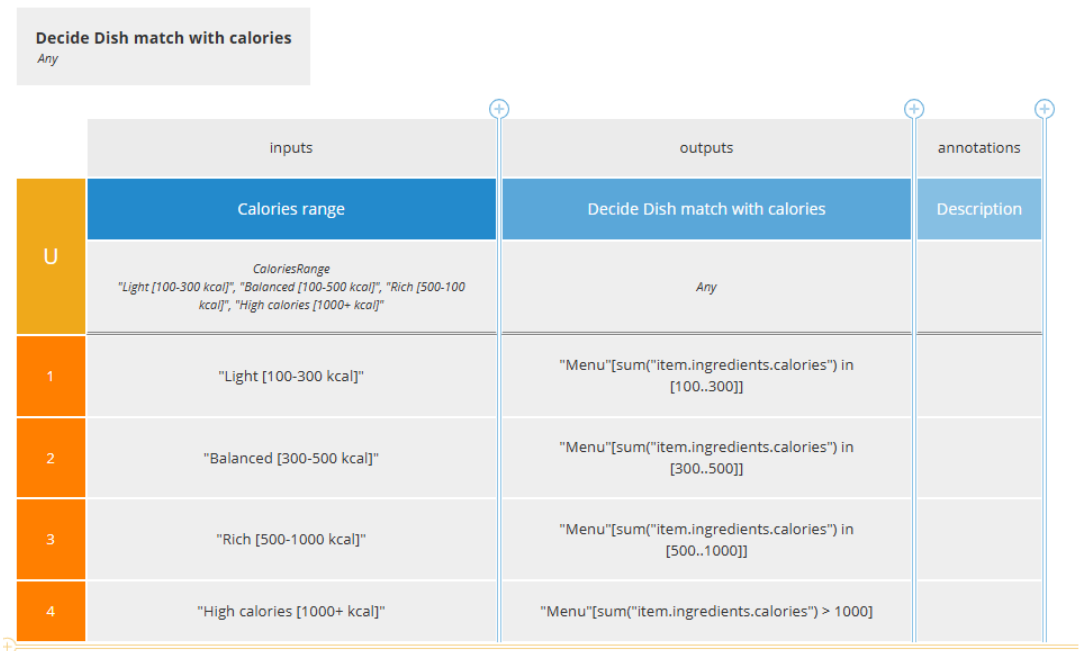


decide on a meal by filtering by diet
Text

	inputs	outputs	annotations
U	Profile	decide on a meal by filtering by diet	Meal's condition
	DietProfile *"carnivor", "vegetarian", "vegan", "fish_based"*	*Text*	
1	"carnivor"	Menu[some i in item.ingredients satisfies i.type = "Meat"]	dish contain meat
2	"vegetarian"	Menu[every i in item.ingredients satisfies not(i.type = "Meat" or i.type = "Fish")]	dish does not contain meat or fish
3	"vegan"	Menu[every i in item.ingredients satisfies not(i.type = "Meat" or i.type = "Fish" or i.type = "Dairy")]	The dish does not contain any ingredients such as meat, fish, eggs, or dairy products.
4	"fish_based"	Menu[some i in item.ingredients satisfies i.type = "Fish"]	dish contain fish

Filtering by Calories

Finally, the system checks the Decide Dish Match with Calories decision. This step ensures that the selected dishes fall within the guest's preferred caloric range. Dishes are evaluated based on their ingredients' calorie content, which is summed up to determine if the dish matches the guest's dietary needs.



Decide Dish match with calories
Any

	inputs	outputs	annotations
U	Calories range	Decide Dish match with calories	Description
	CaloriesRange *"Light [100-300 kcal]", "Balanced [100-500 kcal]", "Rich [500-100 kcal]", "High calories [1000+ kcal]"*	*Any*	
1	"Light [100-300 kcal]"	"Menu"[sum("item.ingredients.calories") in [100..300]]	
2	"Balanced [300-500 kcal]"	"Menu"[sum("item.ingredients.calories") in [300..500]]	
3	"Rich [500-1000 kcal]"	"Menu"[sum("item.ingredients.calories") in [500..1000]]	
4	"High calories [1000+ kcal]"	"Menu"[sum("item.ingredients.calories") > 1000]	

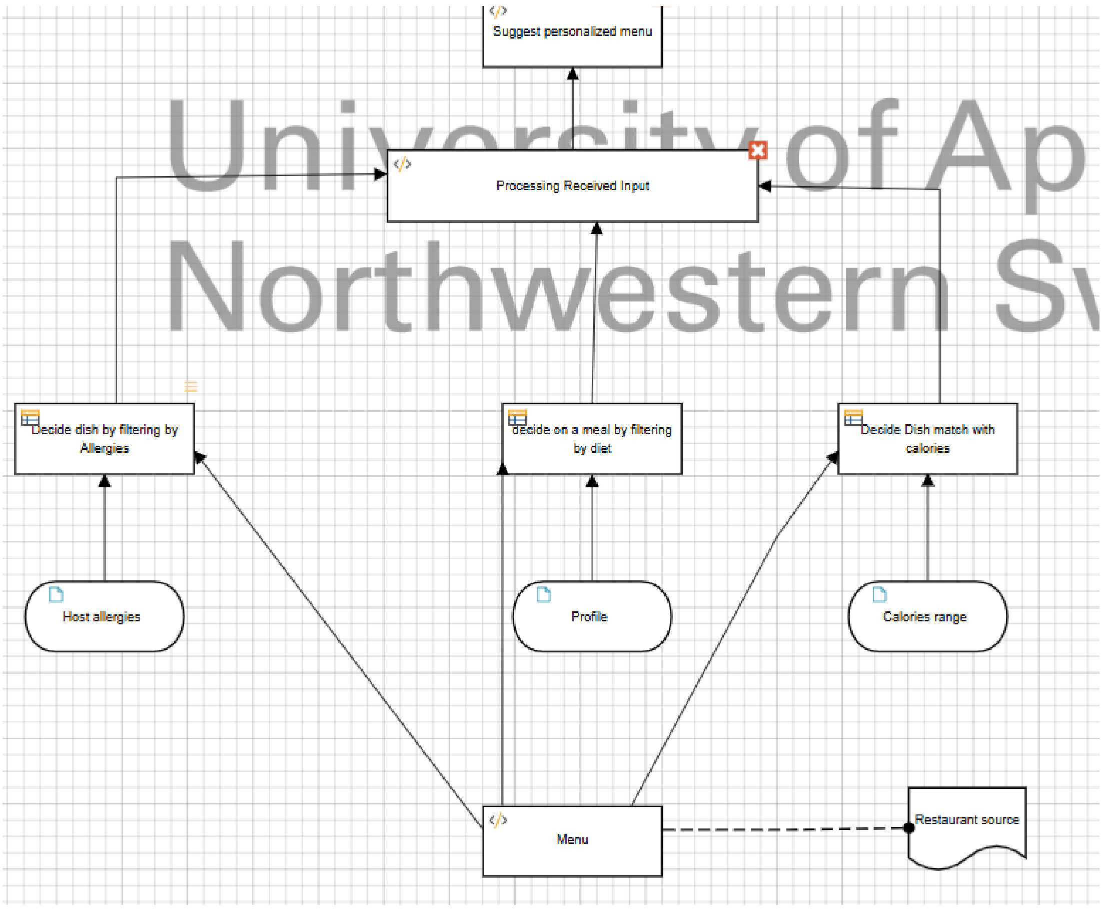
2.1.2 Suggest Personalized Menu

Once all filtering decisions have been made, the system moves to the final decision stage: Suggest Personalized Menu. This decision involves selecting the final set of dishes that meet the guest's allergen, dietary, and caloric constraints. The recommendation logic in this step is as follows:

- The system generates a personalized menu based on the available dishes that match all the filters applied during the previous decision steps.
- The final menu consists of one dish for each course, recommended based on the guest's preferences.
- In the case that the guest does not like the suggested dishes, the system also provides alternative dishes as correlated suggestions.

For example, the system might suggest a dish like "Grilled Salmon" if it meets the guest's dietary and caloric preferences, while also offering alternatives like "Vegetarian Risotto" if the guest prefers a vegetarian option.

The Suggest Personalized Menu decision ensures that the recommendations are tailored specifically to the guest's needs, taking into account all the parameters set by the guest in the input data.


Suggest personalized menu
University of Ap
Northwestern S
Processing Received Input
Decide dish by filtering by Allergies
decide on a meal by filtering by diet
Decide Dish match with calories
Host allergies
Profile
Calories range
Menu
Restaurant source

2.2 Prolog Implementation

The knowledge base is also implemented in Prolog to demonstrate rule-based reasoning. The advantage of Prolog is its declarative nature: instead of writing explicit procedures, I describe facts and rules, and Prolog automatically performs inference to answer queries.

2.3 Prolog

This section outlines the use of Prolog to build a knowledge base that enables personalized menu recommendations based on the guest's profile, known allergies, dietary preferences, and caloric constraints. By asserting facts about guests, ingredients, and dishes, Prolog allows for logical inferences to determine which dishes are suitable for a particular guest.

2.3.1 Facts

The knowledge base starts by asserting each guest's declared profile and known allergen. The predicates ‘guestprofile/1‘ and ‘allergen/1‘ capture the essential information about what a guest eats and what they must avoid.

For example, the following Prolog facts define guest profiles and allergies:

```
guest_profile('Mattia', vegetarian).
guest_profile('Michela', fish_based).

allergen('Mattia', lactose).
allergen('Michela', gluten).
```

This allows us to categorize guests based on their dietary preferences (e.g., vegetarian or pescatarian) and known allergies (e.g., lactose or gluten). These facts form the foundation for filtering dishes during the recommendation process.

Ingredient Categories and Meal Courses

Next, I classify ingredients into broad categories such as meat, fish, vegetable, dairy, etc., using the 'type/1' predicate. The 'course/1' predicate is used to identify the type of dish—whether it is an appetizer, first course, main course, or dessert.

Example Prolog facts for ingredient types and meal courses are:

```
type('Shrimp', 'Fish').
type('Eggs', 'Dairy').
type('Tomato', 'Vegetable').

course('Saltimbocca_alla_Romana', 'Main').
course('Vegetarian_Risotto', 'Main').
course('Tiramisu', 'Dessert').
```

These facts enable the system to categorize dishes based on ingredient type and the course they belong to, allowing for easy filtering based on the guest's preferences.

Ingredient Caloric Content

The next key component in the knowledge base is the caloric content of ingredients. Each ingredient's approximate calorie value is associated using the 'ingredient/2' and 'caloriesingredient/2' predicates. For example:

```
ingredient('Shrimp').
ingredient('Eggs').
ingredient('Tomato').

calories_ingredient('Shrimp', 100).
calories_ingredient('Eggs', 150).
calories_ingredient('Tomato', 20).
```

These facts allow us to calculate the total calorie content of each dish by summing the calories of its ingredients.

Linking Ingredients to Their Categories

The 'typeingredient/2' predicate links individual ingredients to their categories. For example:

```
type_ingredient('Shrimp', 'Fish').
type_ingredient('Eggs', 'Dairy').
type_ingredient('Tomato', 'Vegetable').
```

This association is essential for filtering dishes by dietary profile and grouping ingredients by allergens, such as excluding meat for a vegetarian guest.

Dish Definitions and Ingredients

Once the ingredients and their categories are defined, I enumerate all the available dishes using the ‘dish/1‘ predicate. The relationship between dishes and their ingredients is made explicit using the ‘containsingredient/2‘ predicate.

For example, I can define that Saltimbocca alla Romana contains Veal, Ham, Sage, and Butter:

```
dish('Saltimbocca_alla_Romana').
contains_ingredient('Saltimbocca_alla_Romana', 'Veal').
contains_ingredient('Saltimbocca_alla_Romana', 'Ham').
contains_ingredient('Saltimbocca_alla_Romana', 'Sage').
contains_ingredient('Saltimbocca_alla_Romana', 'Butter').
```

Similarly, I can define the dish Vegetarian Risotto with ingredients like Rice, Tomato, and Spinach:

```
dish('Vegetarian_Risotto').
contains_ingredient('Vegetarian_Risotto', 'Rice').
contains_ingredient('Vegetarian_Risotto', 'Tomato').
contains_ingredient('Vegetarian_Risotto', 'Spinach').
```

These facts are the foundation of the system, which uses the relationship between dishes and ingredients to compute nutritional totals and apply dietary filters.

Dish Course and Allergen Associations

In addition to ingredients, I link each dish to its course type using the ‘dishCourse/2‘ predicate. For example:

```
dish_course('Saltimbocca_alla_Romana', 'Main').
dish_course('Tiramisu', 'Dessert').
```

Finally, I define which allergens are contained in each dish using the ‘containsAllergy/2‘ predicate. For instance, if Saltimbocca alla Romana contains Lactose due to the butter, I would write:

```
contains_allergy('Saltimbocca_alla_Romana', 'Lactose').
contains_allergy('Vegetarian_Risotto', 'None').
```

This information helps in filtering dishes based on the allergens that a guest must avoid.

Prolog-Based Filtering and Menu Generation

With the knowledge base populated with facts, I can now query Prolog to generate a personalized menu based on the guest's dietary profile and allergies. For instance, to recommend a menu for a vegetarian guest who is lactose intolerant, I would query:

```
?- guest_profile('Mattia', vegetarian), allergen('Mattia', lactose),
   dish(Dish), not(contains_allergy(Dish, 'Lactose')),
   contains_ingredient(Dish, Ingredient), type_ingredient(Ingredient, 'Veget
```

This query ensures that the dishes recommended for Mattia are vegetarian, do not contain lactose, and are based on vegetable ingredients.

2.3.2 Rules

Once the knowledge base has been established, the recommendation mechanism is implemented through a concise set of recursive and composite rules. The predicate `dish_calories/2` gathers the caloric values of all the ingredients that compose a dish and computes the total energy content. This value is then used as a foundation for further dietary checks.

Listing 2.1: Prolog rule calculating total calories per dish

```
dish_calories(Dish, Calories) :-
    findall(C, (contains_ingredient(Dish, Ing),
                ingredient_calories(Ing, C)), Cs),
    sum_list(Cs, Calories).
```

The central logic for producing recommendations is encoded in the predicate `recommended/2`. A candidate dish is considered valid if it satisfies all the constraints of the selected dietary profiles, excludes the specified allergens, and remains within the caloric boundaries associated with the profiles. To achieve this, the predicate makes use of helper rules such as `check_profiles/2` and `check_allergies/2`.

Listing 2.2: Main predicate for recommending dishes

```
recommended(Dish, Profiles) :-
    dish_calories(Dish, Cals),
    check_profiles(Profiles, Dish),
    check_allergies(Profiles, Dish),
    within_calories(Profiles, Cals).
```

Profile validation is managed by the composite predicate `check_profiles/2`, which recursively ensures that every profile condition in the guest's list is satisfied by the dish.

Listing 2.3: Recursive check of guest profiles

```
check_profiles([], _).
check_profiles([P|Ps], Dish) :-
    satisfies_profile(P, Dish),
    check_profiles(Ps, Dish).
```

Examples of profile-specific rules include restrictions for vegetarians and vegans.

Listing 2.4: Vegetarian and Vegan profile checks

```
% Vegetarian: no meat
satisfies_profile(vegetarian, Dish) :-
    \+ (contains_ingredient(Dish, Ing),
        ingredient_type(Ing, meat)).

% Vegan: no meat, no dairy
satisfies_profile(vegan, Dish) :-
    \+ (contains_ingredient(Dish, Ing),
        (ingredient_type(Ing, meat);
         ingredient_type(Ing, dairy))).
```

Allergen constraints are handled by `check_allergies/2`, which validates that no unwanted allergens are associated with the dish.

Listing 2.5: Checking allergens in dishes

```
check_allergies([], _).
check_allergies([allergen(A)|As], Dish) :-
    \+ (contains_allergen(Dish, A)),
    check_allergies(As, Dish).
```

Finally, to illustrate the system's functionality, some predefined guest profiles were created. The predicate `suggest_menu/2` makes use of these profiles to return example personalized menus when queried.

Listing 2.6: Suggesting menu options for a given profile

```
suggest_menu(Profile, Menu) :-
    findall(D, recommended(D, Profile), Menu).
```

2.4 Ontology and Knowledge Graph

In this section, I discuss the creation of the knowledge base for the personalized menu system using Protegé and OWL. The primary goal of the ontology is to capture the relationships between dishes, ingredients, guest profiles, allergies, and nutritional information. This ontological model forms the backbone of the recommendation engine, enabling the generation of personalized menus based on dietary preferences and health considerations.

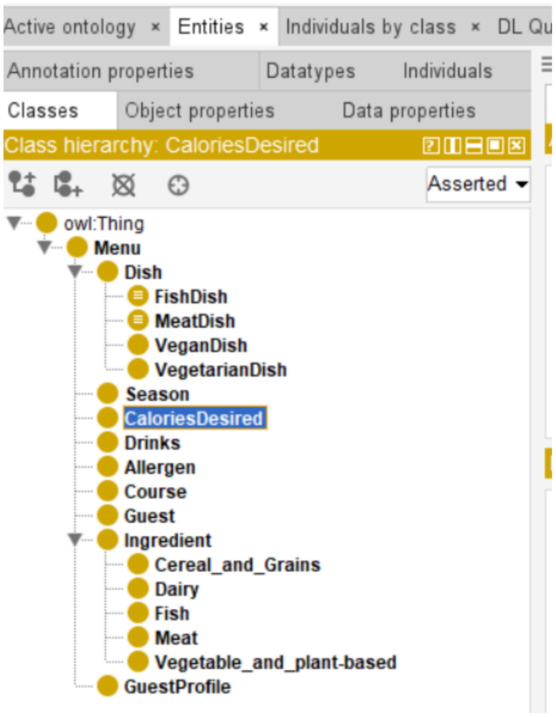
2.4.1 Protégé

The first step in constructing the ontology was defining the main classes that represent the key entities within the menu system. These classes include Dish, Ingredient, Course, allergen, CaloriesDesired, GuestProfile, and Guest. These classes are central to the menu structure, guest preferences, and dietary constraints.

To model the various nutritional categories, I classified ingredients into distinct subclasses, such as Meat, Fish, Dairy, Cereal, and Vegetable. These subclasses were made mutually exclusive using the 'owl:AllDisjointClasses' construct, ensuring that an ingredient can belong to only one category, which aids in accurate filtering during the recommendation process.

For example:

```
Ingredient subclassOf Fish,
owl:AllDisjointClasses(Dairy, Meat, Cereal_and_Grains, Vegetable_and_plant-b
```



```
owl:Thing
  Menu
    Dish
      FishDish
      MeatDish
      VeganDish
      VegetarianDish
    Season
    CaloriesDesired
    Drinks
    Allergen
    Course
    Guest
    Ingredient
      Cereal_and_Grains
      Dairy
      Fish
      Meat
      Vegetable_and_plant-based
    GuestProfile
```

Composite Dish Types and Cardinality Constraints

To represent composite dish types such as MeatDish, FishDish, VegetarianDish, and VeganDish, I applied cardinality restrictions to the 'containsIngredient' object property. This ensures that dishes are classified according to their ingredients. For instance, the MeatDish class is defined as containing at least one ingredient of type Meat:

```
MeatDish equivalentTo some containsIngredient Meat.
```

Similarly, dishes like VegetarianDish are defined by excluding Meat and Dairy ingredients:

```
VegetarianDish equivalentTo some containsIngredient not(Meat, Dairy).
```

This setup ensures that dishes are categorized based on their ingredient composition.

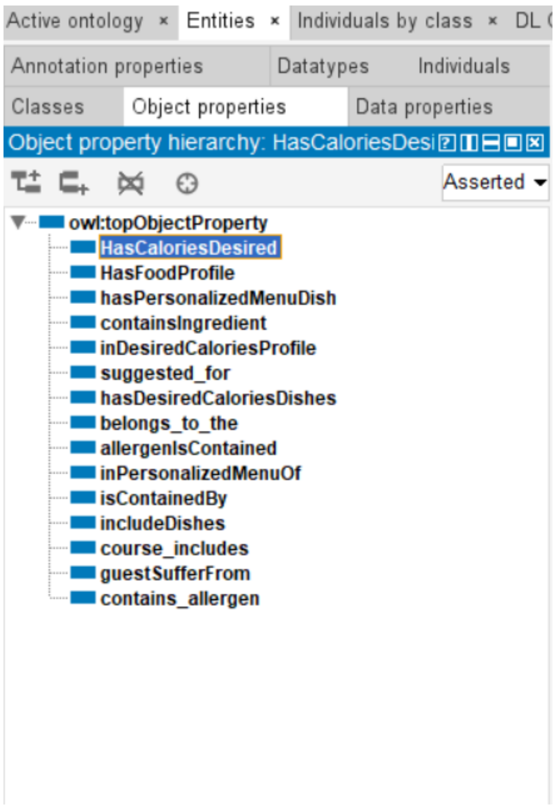
Object Properties and Inverse Relationships

The relationships between dishes and their ingredients, courses, and allergens are modeled using object properties. For example, the object property 'containsIngredient' links a Dish to its Ingredients, while the inverse property 'isContainedBy' links an Ingredient to the Dish it belongs to. Other important object properties include 'containsAllergy' (associating dishes with allergens) and 'belongsToCourse' (associating dishes with their course type, such as appetizer, main, or dessert).

Each object property has a defined domain and range to ensure the relationships are valid. For instance:

```
containsIngredient domain: Dish, range: Ingredient
belongsToCourse domain: Dish, range: Course
containsAllergen domain: Dish, range: Allergen
```

These properties enable Prolog reasoning and ontology classification, helping the system filter dishes based on guest preferences.



owl:topObjectProperty
- HasCaloriesDesired
- HasFoodProfile
- hasPersonalizedMenuDish
- containsIngredient
- inDesiredCaloriesProfile
- suggested_for
- hasDesiredCaloriesDishes
- belongs_to_the
- allergenIsContained
- inPersonalizedMenuOf
- isContainedBy
- includeDishes
- course_includes
- guestSufferFrom
- contains_allergen

Quantitative Information: Calories and Caloric Preferences

In the ontology, I associate ingredients with their caloric values using datatype properties. Each Ingredient carries a mandatory ‘hasCalories‘ property, while Dish instances expose a precomputed ‘hasTotalCalories‘ value.

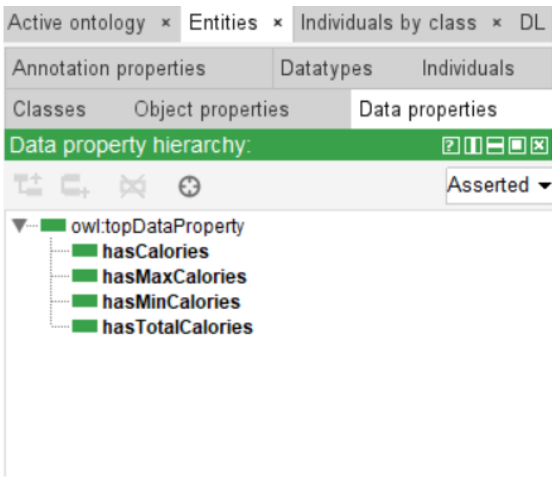
For example:

```
hasCalories('Veal', 110).
hasCalories('Butter', 325).
hasCalories('Spinach', 20).
```

To represent the caloric ranges requested by the guest, such as Light, Moderate, Hearty, or High, I defined the ‘CaloriesDesired‘ class. Each instance of this class represents a specific calorie range with ‘hasMinCalories‘ and ‘hasMaxCalories‘ properties, closing the numeric intervals for calorie limits:

```
CaloriesDesired('Light') hasMinCalories 100 hasMaxCalories 300.
CaloriesDesired('Balanced') hasMinCalories 300 hasMaxCalories 500.
CaloriesDesired('Rich') hasMinCalories 500 hasMaxCalories 1000.
CaloriesDesired('High') hasMinCalories 1000.
```

By explicitly defining these ranges, the ontology allows for precise control over calorie filtering during the recommendation process.



owl:topDataProperty
hasCalories
hasMaxCalories
hasMinCalories
hasTotalCalories

Instantiating Dishes and Guest Profiles

With the classes, object properties, and axioms in place, the next step is to instantiate the specific dishes and guest profiles. Each dish is linked to its ingredients, course type, and allergens using facts. For example:

```
dish('Saltimbocca_alla_Romana').
containsIngredient('Saltimbocca_alla_Romana', 'Veal').
containsIngredient('Saltimbocca_alla_Romana', 'Butter').
containsIngredient('Saltimbocca_alla_Romana', 'Sage').
belongsToCourse('Saltimbocca_alla_Romana', 'Main').
containsAllergy('Saltimbocca_alla_Romana', 'Lactose').
```

Guest profiles are similarly instantiated, allowing for personalized filtering based on each guest's dietary preferences and allergies:

```
guestProfile('Mattia', vegetarian).
guestProfile('Michela', fish_based).
```

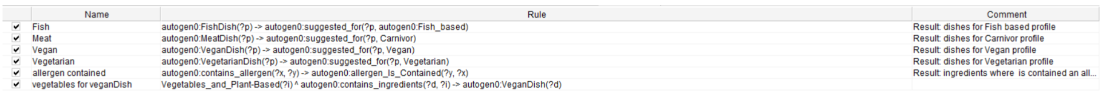
This allows the system to query the ontology and retrieve personalized menu recommendations.

Reasoning and Classification

Once the ontology is populated with individuals, I use HermiT (the OWL reasoner) to classify the entities and ensure that all axioms and relationships are logically consistent. For example, after reasoning, the reasoner will automatically classify Grilled Salmon under both Dish and FishDish, based on its ingredients.

2.4.2 SWRL Rules

This section describes the SWRL (Semantic Web Rule Language) queries used to generate personalized menu recommendations based on guest profiles. The queries play an essential role in filtering dishes according to specific dietary preferences, allergies, and nutritional constraints. Below, I explain each rule and its purpose in detail, describing how it impacts the menu recommendation process.



	Name	Rule	Comment
✓	Fish	autogen0:FishDish(?p) -> autogen0:suggested_for(?p, autogen0:Fish_based)	Result: dishes for Fish based profile
✓	Meat	autogen0:MeatDish(?p) -> autogen0:suggested_for(?p, Carnivor)	Result: dishes for Carnivor profile
✓	Vegan	autogen0:VeganDish(?p) -> autogen0:suggested_for(?p, Vegan)	Result: dishes for Vegan profile
✓	Vegetarian	autogen0:VegetarianDish(?p) -> autogen0:suggested_for(?p, Vegetarian)	Result: dishes for Vegetarian profile
✓	allergen contained	autogen0:contains_allergen(?x, ?y) -> autogen0:allergen_Is_Contained(?y, ?x)	Result: ingredients where is contained an all...
✓	vegetables for veganDish	Vegetables_and_Plant-Based(?i) ^ autogen0:contains_ingredients(?d, ?i) -> autogen0:VeganDish(?d)	

Rules for Dish Profiles

Fish: The following rule suggests dishes that are categorized as FishDish for guests who have a Fish-based diet. It checks whether the dish belongs to the FishDish category and then associates it with the Fish-based profile of the guest. This rule ensures that guests who prefer fish as their primary source of protein will only be recommended dishes that meet this requirement.

```
FishDish(?p) -> suggested_for(?p, Fish_based)
```

Explanation: This rule ensures that all FishDish types are linked to the Fish-based diet profile. Thus, if the guest is classified under a fish-based diet, they will be recommended dishes like Grilled Salmon or Fish Tacos that fit this profile.

Meat: This rule filters out dishes that are not suitable for guests following a Carnivorous diet, specifically recommending MeatDish dishes. It associates any dish containing meat with the Carnivorous profile.

```
MeatDish(?p) -> suggested_for(?p, Carnivor)
```

Explanation: Guests who have a Carnivorous diet will only receive suggestions for MeatDish types, ensuring that dishes such as Steak or Roast Chicken are suggested, while vegetarian or vegan dishes are excluded.

Vegan: This rule ensures that dishes suitable for a Vegan diet are recommended. It checks if the dish falls under the VeganDish category, meaning the dish contains only plant-based ingredients, free from meat, dairy, and other animal products.

```
VeganDish(?p) -> suggested_for(?p, Vegan)
```

Explanation: This rule is designed to suggest dishes like Vegan Salad or Vegetable Stir-fry to guests who follow a vegan diet. The dishes must strictly exclude any animal-derived ingredients like meat, dairy, or eggs.

Vegetarian: Similar to the vegan rule, this query suggests dishes classified as VegetarianDish for guests who follow a Vegetarian diet. It filters out any dishes containing meat, ensuring only plant-based or dairy-inclusive dishes are recommended.

```
VegetarianDish(?p) ->suggested_for(?p,Vegetarian)
```

Explanation: This rule ensures that guests who prefer vegetarian meals are recommended dishes like Vegetarian Risotto or Vegetable Curry, where the focus is on plant-based ingredients with no meat involved.

Rules for Allergens and Ingredient Types

Allergen Contained: This rule checks whether a dish contains a specific allergen. It links allergens to dishes, flagging dishes that contain ingredients that guests need to avoid due to allergies (such as Lactose, Gluten, or Nuts). The ‘containsallergen/2‘ predicate is triggered when the dish contains an ingredient that is flagged with an allergen.

```
contains_allergen(?x, ?y) ->allergen_Is_Contained(?x, ?y)
```

Explanation: If a dish contains an allergen that a guest is sensitive to (e.g., lactose or gluten), this rule ensures that the dish is not recommended to that guest. This check prevents cross-contamination or potential health issues.

Vegetables for VeganDish: This rule ensures that dishes identified as VeganDish include only plant-based ingredients, such as vegetables and plant-based proteins. It filters out dishes that contain non-plant-based ingredients like meat or dairy, ensuring that the dish is strictly vegan. This rule is especially important when ensuring that vegan guests receive only plant-based options.

```
Vegetables_and_Plant-Based(?i) ^ contains_ingredients(?d, ?i)
->VeganDish(?d)
```

Explanation: This rule guarantees that VeganDish dishes contain only vegetable-based ingredients and are free from animal products. For example, a dish like Vegan Tacos that uses plant-based ingredients like tofu or vegetables will be selected, while meat-based dishes will be excluded.

2.4.3 SPARQL Queries

This section includes a set of SPARQL queries for retrieving and manipulating data stored in the RDF graph related to personalized menu recommendations. These queries are designed to interact with the RDF model defined in the ontology, specifically handling dishes, ingredients, allergens, guest profiles, and dietary requirements.

Query 1: Retrieve Dishes for a Specific Food Profile

This query retrieves all dishes that match a specific food profile (e.g., Vegetarian, Vegan). It filters dishes that correspond to a particular dietary preference.

```
PREFIX : <http://www.semanticweb.org/kebi.marinangelidiego#>

SELECT ?dish ?course ?calories
WHERE {
  ?dish a :Dish ;
        :hasTotalCalories ?calories ;
        :belongs_to_the ?course ;
        :suggested_for ?foodProfile .

  VALUES ?foodProfile { :Vegetarian }
}
```

Explanation: This query returns all dishes that are classified under the Vegetarian profile. The ‘:suggestedFor‘ property links the dish to the guest’s profile.

Query 2: Retrieve Dishes with Specific Ingredients

This query retrieves dishes containing a specific type of ingredient, such as Meat or Fish. It helps filter dishes based on dietary restrictions (e.g., fishBased, carnivorous).

```
PREFIX : <http://www.semanticweb.org/kebi.marinangelidiego#>

SELECT ?dish
WHERE {
  ?dish :containsIngredient ?ingredient .
  ?ingredient :type :Meat
}
```

Explanation: This query selects all dishes that contain Meat as an ingredient. It uses the ‘:containsIngredient‘ relationship to find dishes with specific ingredients.

Query 3: Retrieve Dishes Avoiding a Specific Allergen

This query is designed to retrieve dishes that do not contain a specified allergen, like Lactose or Gluten.

```
PREFIX : <http://www.semanticweb.org/kebi.marinangelidiego#>

SELECT ?dish
WHERE {
  ?dish :containsAllergy ?allergen .
  FILTER NOT EXISTS { ?allergen :name "Lactose" }
}
```

Explanation: This query finds all dishes that do not contain Lactose as an allergen. It uses a ‘FILTER NOT EXISTS‘ clause to exclude dishes with the specified allergen.

Query 4: Retrieve Dishes by Caloric Range

This query retrieves dishes that fall within a certain caloric range. For example, it can retrieve dishes that are Balanced in calories.

```
PREFIX : <http://www.semanticweb.org/kebi.marinangelidiego#>

SELECT ?dish ?calories
WHERE {
  ?dish :hasTotalCalories ?calories .
  FILTER (?calories >= 200 && ?calories <= 700)
}
```

Explanation: This query selects dishes with a total caloric value between 200 and 700 kcal. It uses the ‘:hasTotalCalories‘ property to filter the dishes based on their caloric content.

Query 5: Retrieve Dishes Avoiding Multiple Allergens

This query retrieves dishes that avoid multiple allergens. It is especially useful for guests with complex dietary needs.

```
PREFIX : <http://www.semanticweb.org/kebi.marinangelidiego#>

SELECT ?dish
WHERE {
  ?dish :containsAllergy ?allergen1 .
  ?dish :containsAllergy ?allergen2 .
  FILTER NOT EXISTS { ?allergen1 :name "Gluten" }
  FILTER NOT EXISTS { ?allergen2 :name "Dairy" }
}
```

Explanation: This query finds dishes that do not contain both Gluten and Dairy as allergens. It uses two ‘FILTER NOT EXISTS‘ clauses to exclude dishes with these allergens.

Query 6: Retrieve Dishes by Course Type

This query retrieves dishes categorized under a specific course type (e.g., Main Course, Dessert).

```
PREFIX : <http://www.semanticweb.org/kebi.marinangelidiego#>

SELECT ?dish
WHERE {
  ?dish :belongs_to_the :MainCourse
}
```

Explanation: This query selects all dishes that belong to the Main Course category. The ‘:belongs$_{t}o_{t}he‘propertylinksdishestotheirrespectivecoursetypes.$

Query 7: Retrieve Dishes for a Specific Guest Profile

This query retrieves all dishes suitable for a guest with a specific dietary profile (e.g., Vegan, FishBased).

```
PREFIX : <http://www.semanticweb.org/kebi.marinangelidiego#>

SELECT ?dish
WHERE {
  ?dish :suggested_for :Vegan
}
```

Explanation: This query retrieves dishes that are suitable for a Vegan guest profile. It filters dishes by the ‘:suggestedFor‘ relationship, which connects dishes to guest profiles.

2.4.4 SHACL Constraints

In this section, I define SHACL (Shapes Constraint Language) constraints that validate the RDF data of the personalized menu system. SHACL allows us to enforce structural integrity in the RDF graph, ensuring that the data conforms to the required rules, such as relationships between dishes and ingredients, guest profiles, allergens, and calories.

The SHACL constraints are defined using shapes, which describe the structure that specific resources must adhere to. A shape is essentially a template for validating data and ensuring that it meets the expected conditions.

SHACL Constraints for Dishes

```
@prefix :      <http://www.semanticweb.org/kebi.marinangelidiego#> .
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix sh:    <http://www.w3.org/ns/shacl#> .
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:   <http://www.w3.org/2002/07/owl#> .
ex:DishShape a sh:NodeShape ;
    sh:targetClass ex:Dish ;
    sh:property [
        sh:path ex:hasName ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
    ] ;
    sh:property [
        sh:path ex:containsIngredient ;
        sh:class ex:Ingredient ;
        sh:minCount 1 ;
    ] ;
    sh:property [
        sh:path ex:belongsToCourse ;
        sh:class ex:Course ;
        sh:minCount 1 ;
    ] .
```

This shape ensures that each dish has a valid name, contains at least one ingredient, and is assigned to at least one course.

SHACL Constraints for Ingredients

```
ex:IngredientShape a sh:NodeShape ;
    sh:targetClass ex:Ingredient ;
    sh:property [
        sh:path ex:type ;
        sh:in (ex:Meat ex:Fish ex:Dairy ex:Vegetable ex:Cereal) ;
        sh:minCount 1 ;
    ] ;
    sh:property [
        sh:path ex:hasCalories ;
        sh:datatype xsd:integer ;
        sh:minCount 1 ;
    ] .
```

This shape ensures that each ingredient is classified under one of the predefined types and has a defined calorie count.

SHACL Constraints for Guest Profiles

```
ex:GuestProfileShape a sh:NodeShape ;
    sh:targetClass ex:GuestProfile ;
    sh:property [
        sh:path ex:profileType ;
        sh:in (ex:Vegetarian ex:Vegan ex:Pescatarian) ;
        sh:minCount 1 ;
    ] ;
    sh:property [
        sh:path ex:canConsumeDish ;
        sh:class ex:Dish ;
        sh:minCount 1 ;
    ] .
```

This shape ensures that each guest profile is assigned to a specific profile type and is linked to at least one dish they can consume.

SHACL Constraints for Allergens

```
ex:AllergenShape a sh:NodeShape ;
    sh:targetClass ex:allergen ;
    sh:property [
        sh:path ex:containsAllergen ;
        sh:class ex:AllergyType ;
        sh:minCount 0 ;  # allergen is optional
    ] ;
    sh:property [
        sh:path ex:linkedToDish ;
        sh:class ex:Dish ;
        sh:minCount 1 ;
    ] .
```

This shape ensures that allergen resources are properly linked to allergens and associated with dishes that may contain them.

SHACL Constraints for Caloric Ranges

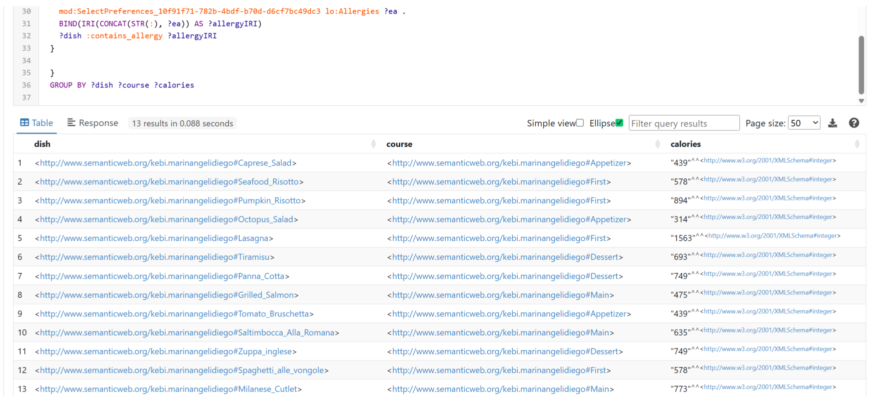
```
ex:CaloriesDesiredShape a sh:NodeShape ;
    sh:targetClass ex:CaloriesDesired ;
    sh:property [
        sh:path ex:hasMinCalories ;
        sh:datatype xsd:int ;
        sh:minCount 1 ;
    ] ;
    sh:property [
        sh:path ex:hasMaxCalories ;
        sh:datatype xsd:int ;
        sh:minCount 1 ;
    ] ;
    sh:property [
        sh:path ex:belongsToCaloriesRange ;
        sh:class ex:Dish ;
        sh:minCount 1 ;
    ] .
```

This shape ensures that each caloric range has both minimum and maximum values and is linked to one or more dishes.

3. Agile BPMN Meta-Modelling

In my effort to enhance classical business process modeling with semantic precision, I have employed an agile meta-modelling methodology that integrates BPMN 2.0 constructs with an OWL-based knowledge graph, alongside SWRL rules and SPARQL queries. This strategy recognizes that the domain of personalized menu recommendation is highly dynamic: new dietary profiles continuously arise, ingredients vary seasonally, and allergen data must be kept accurate.

The core of our solution is the *Personalized Menu Ontology*, developed in Turtle syntax. It includes classes such as `:Dish`, `:Ingredient`, `:Course`, and `:Profile`. Nutritional information, such as caloric values, is represented as data properties, while user preferences and dietary restrictions—including vegetarian choices or lactose and gluten intolerance—are modeled as object properties linking dishes to profile instances. Once the ontology is loaded into Apache Jena Fuseki, available at `http://localhost:3030`, SPARQL queries can retrieve candidate dishes that match a guest's `lo:GuestProfile` and `lo:CaloriesProfile`, while automatically excluding dishes that violate specified allergies. Consequently, the SPARQL endpoint serves not just as a repository but also as an active reasoning engine, returning results such as Caprese Salad, Tomato Bruschetta, and Tiramisu with their caloric content, as shown in Figure 3.1.



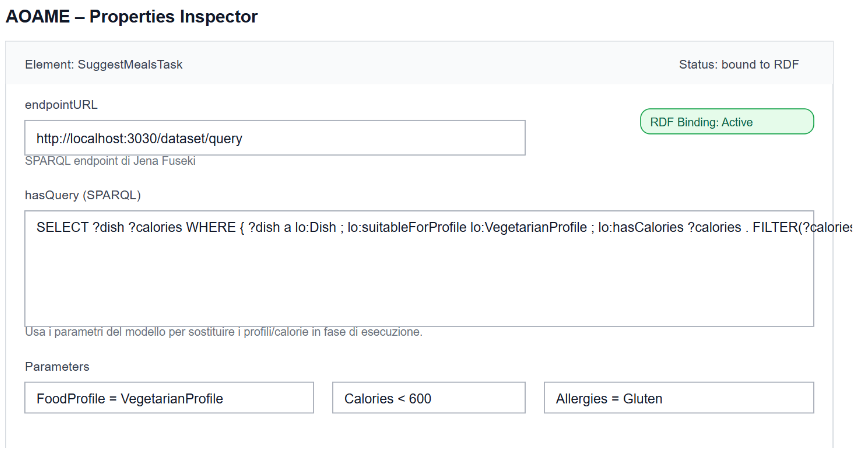
```
    mod:SelectPreferences_10f91f71-782b-4bdf-b70d-d6cf7bc49dc3 lo:Allergies ?ea .
    BIND(IRI(CONCAT(STR(:), ?ea)) AS ?allergyIRI)
    ?dish :contains_allergy ?allergyIRI
  }

  }
  GROUP BY ?dish ?course ?calories
```

	dish	course	calories
1	<http://www.semanticweb.org/kebi.marinangelidiego#Caprese_Salad>	<http://www.semanticweb.org/kebi.marinangelidiego#Appetizer>	"439"^^<http://www.w3.org/2001/XMLSchema#integer>
2	<http://www.semanticweb.org/kebi.marinangelidiego#Seafood_Risotto>	<http://www.semanticweb.org/kebi.marinangelidiego#First>	"578"^^<http://www.w3.org/2001/XMLSchema#integer>
3	<http://www.semanticweb.org/kebi.marinangelidiego#Pumpkin_Risotto>	<http://www.semanticweb.org/kebi.marinangelidiego#First>	"894"^^<http://www.w3.org/2001/XMLSchema#integer>
4	<http://www.semanticweb.org/kebi.marinangelidiego#Octopus_Salad>	<http://www.semanticweb.org/kebi.marinangelidiego#Appetizer>	"314"^^<http://www.w3.org/2001/XMLSchema#integer>
5	<http://www.semanticweb.org/kebi.marinangelidiego#Lasagna>	<http://www.semanticweb.org/kebi.marinangelidiego#First>	"1563"^^<http://www.w3.org/2001/XMLSchema#integer>
6	<http://www.semanticweb.org/kebi.marinangelidiego#Tiramisu>	<http://www.semanticweb.org/kebi.marinangelidiego#Dessert>	"693"^^<http://www.w3.org/2001/XMLSchema#integer>
7	<http://www.semanticweb.org/kebi.marinangelidiego#Panna_Cotta>	<http://www.semanticweb.org/kebi.marinangelidiego#Dessert>	"749"^^<http://www.w3.org/2001/XMLSchema#integer>
8	<http://www.semanticweb.org/kebi.marinangelidiego#Grilled_Salmon>	<http://www.semanticweb.org/kebi.marinangelidiego#Main>	"475"^^<http://www.w3.org/2001/XMLSchema#integer>
9	<http://www.semanticweb.org/kebi.marinangelidiego#Tomato_Bruschetta>	<http://www.semanticweb.org/kebi.marinangelidiego#Appetizer>	"439"^^<http://www.w3.org/2001/XMLSchema#integer>
10	<http://www.semanticweb.org/kebi.marinangelidiego#Saltimbocca_Alla_Romana>	<http://www.semanticweb.org/kebi.marinangelidiego#Main>	"635"^^<http://www.w3.org/2001/XMLSchema#integer>
11	<http://www.semanticweb.org/kebi.marinangelidiego#Zuppa_inglese>	<http://www.semanticweb.org/kebi.marinangelidiego#Dessert>	"749"^^<http://www.w3.org/2001/XMLSchema#integer>
12	<http://www.semanticweb.org/kebi.marinangelidiego#Spaghetti_alle_vongole>	<http://www.semanticweb.org/kebi.marinangelidiego#First>	"578"^^<http://www.w3.org/2001/XMLSchema#integer>
13	<http://www.semanticweb.org/kebi.marinangelidiego#Milanese_Cutlet>	<http://www.semanticweb.org/kebi.marinangelidiego#Main>	"773"^^<http://www.w3.org/2001/XMLSchema#integer>

Figure 3.1: SPARQL query results in Jena Fuseki (`localhost:3030`)

Building on this semantic foundation, the Agile and Ontology-Aided Meta-Modelling Environment (AOAME), accessible at `http://localhost:4200`, provides an extended BPMN palette where the generic `Task` element is specialized into `:SuggestMealsTask`. This new task type includes two custom properties: `:hasQuery` and `:endpointURL`, which store the SPARQL SELECT query and the Fuseki service URI, respectively. The *Guest* pool features a distinct task color, visually distinguishing semantic tasks from standard ones and signaling to the restaurant manager when semantic reasoning is invoked. Such visual cues, combined with the meta-model, simplify the design and comprehension of personalized recommendation workflows.

A key aspect of our implementation is the ability to parameterize process elements through model-level attributes. As illustrated in Figure 3.2, users can set values for `FoodProfile`, `CaloriesProfile`, and `Allergies` directly within the task properties inspector for elements such as `ElaborateSuggestedMenu`. AOAME propagates these parameters to the corresponding RDF triples, updating the Fuseki dataset in real time. This bidirectional binding ensures that the process model remains the authoritative source of truth for both workflow logic and data semantics.



AOAME – Properties Inspector

Element: SuggestMealsTask

Status: bound to RDF

endpointURL

http://localhost:3030/dataset/query

SPARQL endpoint di Jena Fuseki

RDF Binding: Active

hasQuery (SPARQL)

```
SELECT ?dish ?calories WHERE { ?dish a lo:Dish ; lo:suitableForProfile lo:VegetarianProfile ; lo:hasCalories ?calories . FILTER(?calorie
```

Usa i parametri del modello per sostituire i profili/calorie in fase di esecuzione.

Parameters

FoodProfile = VegetarianProfile

Calories < 600

Allergies = Gluten

Figure 3.2: Task attributes in AOAME linking process parameters to ontology instances

At runtime, when a guest interacts with the "Select Menu Preferences" task, AOAME converts the selected options into SWRL rule inputs and SPARQL query parameters. Execution of the specialized `SuggestMealsTask` triggers an HTTP GET request to the Fuseki SPARQL endpoint. The JSON response is parsed by the AOAME engine, which creates BPMN data objects representing `:Dish` individuals with associated course and caloric information. These data objects are then automatically connected to the subsequent "Show Suggested Menu" task, allowing the interface to display a customized list of dishes.

From an agile development perspective, this architecture decouples the semantic backend from the BPMN process notation. Ontology engineers can refine class hierarchies, add new dietary profiles, or introduce additional reasoning rules without altering the BPMN runtime. Likewise, business analysts can modify the workflow—adding approval gateways, logging tasks, or alternative recommendation strategies—without affecting the integrity of the knowledge graph. This interplay enables rapid iterations and continuous delivery, hallmarks of agile methodology.

4. Conclusions

Throughout my work on developing a personalized menu recommendation system, I have extensively explored knowledge-based solutions that leverage ontologies, semantic rules, and SPARQL queries. Reflecting on this experience, I can identify both significant advantages and certain limitations inherent in this approach.

4.1 Advantages

One of the main benefits I have observed is the **expressive power** of knowledge-based systems. By formally representing dishes, ingredients, dietary profiles, and nutritional constraints, I am able to capture complex relationships that would be difficult to encode using traditional procedural programming. This allows for a highly flexible recommendation process that can adapt dynamically to new user preferences, dietary restrictions, and seasonal ingredient variations.

Another advantage is **reusability and maintainability**. Once the ontology and associated rules are defined, they can be reused across different processes and applications. For example, the same dietary ontology could support not only menu recommendations but also inventory management, nutritional analysis, or personalized marketing campaigns. This modularity reduces the need for repetitive coding and enables rapid adaptation to changing requirements.

I also appreciated the **transparency and explainability** provided by semantic reasoning. When a recommendation is generated, I can trace back the decision to specific rules and ontology instances, which increases trust in the system from both developers and end-users. This is particularly valuable in domains where compliance and accuracy are critical, such as health-conscious meal planning.

4.2 Disadvantages

Despite these advantages, I have also encountered several challenges. First, the **initial effort and complexity** required to build an ontology and define rules are significant. Designing a comprehensive model that covers all possible ingredients, dietary profiles, and exceptions demands deep domain knowledge and meticulous attention to detail.

Another limitation is the **performance overhead**. Querying large ontologies with complex SPARQL queries can be slower than conventional database lookups, especially when reasoning engines are involved. While this may not be critical for small-scale applications, it could become a bottleneck in real-time or high-volume systems.

I have also noticed that **rigidity in rule-based reasoning** can sometimes restrict flexibility. Knowledge-based systems perform exceptionally well when rules are clearly defined, but they may struggle with ambiguous or incomplete data. In such cases, purely statistical or machine learning approaches could complement the system, providing robustness where rules alone are insufficient.

4.3 Personal Reflection

Overall, working with knowledge-based solutions has been an enriching experience. I have learned to appreciate the power of semantic modeling, while also recognizing the need to balance precision with practicality. In my view, these systems are highly valuable in domains that require structured knowledge and clear decision-making, but they may need to be integrated with other approaches to handle uncertainty, scalability, and performance constraints effectively.

In conclusion, I believe that knowledge-based solutions offer a compelling framework for designing intelligent and explainable systems, such as personalized menu recommendation. By carefully managing their complexity and combining them with complementary techniques, I can harness their strengths while mitigating their limitations, ultimately delivering a more reliable and adaptable solution.