University of Camerino

School of Sciences and Technologies

Master degree in Computer Science (Class LM-18)

Personalized Menu

KEBI - Project

Group members

Samuele Pirani - 131034
Matteo Machella - 131725

Supervisors

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

A.A. 2024/2025

Contents

List of Figures

1. Introduction

The intent of this document is to present, in a unified and coherent narrative, the full development lifecycle of our *Personalized Menu* project for the course *Knowledge Engineering and Business Intelligence*. Throughout these pages, we articulate not only the functional goals and high-level methodology but also the specific technologies and modelling paradigms employed to achieve a truly adaptive restaurant menu recommendation system.

In Section 1.1 we provide a concise overview of the system's requirements and the domain problem—namely, how to filter and present an Italian restaurant's offerings in accordance with a guest's dietary preferences, caloric constraints and allergenic restrictions. Section 1.2 then enumerates the sequence of tasks required to fulfil these objectives, from crafting decision tables to implementing logical inference in Prolog, from building an OWL knowledge graph enriched with SWRL, SPARQL and SHACL, to extending BPMN 2.0 via an agile, ontology-aided meta-modelling environment.

Chapter 2 delivers a detailed exposition of our *Knowledge-Based Solution*, beginning with the declarative DMN decision tables (Section 2.1), continuing with the Prolog facts and rules (Section 2.2), and culminating in the ontology-centric approach realized in Protegé and validated through SHACL (Sections2.3.1 through 2.3.4). Chapter3 then demonstrates how we repurpose and extend this knowledge graph within a BPMN 2.0 workflow, using AOAME and Jena Fuseki to execute live SPARQL queries from specialized `SuggestMealsTask` elements.

Finally, Chapter 4 synthesizes our findings through two distinct, first-person reflections: Matteo Machella's (Section 4.1) emphasis on the trade-offs between stakeholder-friendly formalisms and expressive power, and Samuele Pirani's (Section 4.2) assessment of tool usability versus theoretical elegance. Together, these perspectives close the loop on our exploration, offering practical insights and recommendations for future work in agile, ontology-driven process engineering.

1.1 Project Description

Many restaurants have their menus digitized. Guests can scan a QR code and have the menu presented on their smartphones. A disadvantage is that the screen is very small and it is difficult to get an overview, in particular if the menu is large. However, some guests cannot or do not want every meal, e.g. vegetarians or guests with an allergy. Instead of showing all the meals offered, it would be preferable to show only those meals the guest prefers.

The objective of the project is to represent the knowledge about meals and guest preferences and to create a system that allows for the selection of meals that match the guest preferences.

The knowledge base shall contain information about typical meals of an Italian restaurant, e.g. pizza, pasta, and main dishes. The meals consist of ingredients. There are different types of ingredients, such as meat, vegetables, fruits, or dairy. For each ingredient, there is information on calories.

Guests can be carnivores, vegetarians, calorie conscious, or suffer from allergies, e.g. lactose or gluten intolerance.

1.2 Task List

The following list describes all the tasks to tackle in order to complete the final project:

1. Create different knowledge-based solutions for recommending food depending on the profile of a guest (carnivores, vegetarians, calorie-conscious, suffering from allergies, etc.) using the following representation languages:
   - Decision tables (including DRD with sub-decisions and corresponding decision tables);
   - Prolog (including facts and rules);
   - Knowledge graph/Ontology (including rules in SWRL, queries in SPARQL and SHACL shapes);
2. Agile and ontology-based meta-modeling: adapt BPMN 2.0 to suggest the meals for a given customer. For this, you can re-use or extend the knowledge graph/ontology created in the previous task. One option that you have is to specify the class BPMN Task with a new class and add additional properties, similar to what we have done in class with the Business Process as a Service case. Think of a new graphical notation for the new modeling element, which could be easy to understand for the restaurant manager. Use the triple store interface (Jena Fuseki) to fire the query result.

2. Knowledge Based Solution

In this chapter, the knowledge-based solution realized for the personalized-menu is presented in detail. As described in Section 1.1, the initial interpretation relied on Decision Tables (see Section 2.1) to filter the restaurant's dishes based on guests' preferences.

After the definition of Decision Tables, we provided an implementation of the Prolog script to create the recommendation system for the personalized menu (see Section 2.2).

In addition to that, we created an ontology based on Protegé powered by SPARQL querys and SHACL validation (see Section 2.3.1).

2.1 Decision Tables

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

Figure 2.1 illustrates this dish list. Once this central decision is established, the system is designed to further refine the menu through specialized decision modules that focus on three main aspects: filtering dishes based on allergenic content, tailoring the selection to match guest dietary preferences, and aligning dishes with caloric requirements. The module that handles allergenic content takes into account common allergens such as lactose, eggs, peanuts, tree nuts, fish, shellfish, mollusks, soy, and gluten. These allergens have been identified as the most significant due to their frequent occurrence and potential to affect a considerable portion of the clientele. In parallel, the solution considers the guest's dietary profile, taking into account preferences that range from carnivorous diets to vegetarian, vegan, and fish-based diets, thereby capturing a broad spectrum of culinary tastes. In addition, the system evaluates the caloric aspect by distinguishing between various nutritional levels. It categorizes dishes in relation to their energy content by defining ranges such as Light or Diet Friendly (200 to 400 kcal), Moderate or Balanced (200 to 700 kcal), Hearty or Energy Rich (200 to 1000 kcal), and High Calorie (above 1000 kcal). This structured approach ensures that a wide range of input data is available for the decision-making process, which ultimately enhances the precision of the personalized recommendations. The implementation of these decision processes is illustrated by several figures.

Figure 2.2 presents the decision table used to filter out dishes containing allergens, ensuring that no item contradicts the specific health or dietary constraints of the guest.

Similarly, Figure 2.3 shows the decision table that matches dishes to the guest's declared profile, and Figure 2.4 details the decision table for aligning the dishes with the guest's caloric profile. These sub-decisions operate sequentially and their outcomes are subsequently merged by a main decision module known as *Filter Remaining Menu*.

```
 { "name": "Tomato Bruschetta",
  "ingredients": [
    { "name": "Bread", "type": "Cereal","calories": 250 },
    { "name": "Tomato", "type": "Vegetable", "calories": 18 },
    { "name": "Garlic", "type": "Vegetable", "calories": 149 },
    { "name": "Basil", "type": "Vegetable", "calories": 22 }
  ],
  "course": "Appetizer",
  "allergens": ["gluten"]
},
{ "name": "Seafood Risotto",
  "ingredients": [
    { "name": "Rice", "type": "Cereal","calories": 130 },
    { "name": "Shrimp", "type": "Fish", "calories": 85 },
    { "name": "Squid", "type": "Fish", "calories": 92 },
    { "name": "Mussels", "type": "Fish", "calories": 86 },
    { "name": "Garlic", "type": "Vegetable", "calories": 149 },
    { "name": "Parlsey", "type": "Vegetable", "calories": 36 }
  ],
  "course": "First",
  "allergens": ["shellfish", "mollusks"]
},
  { "name": "Saltimbocca alla Romana",
  "ingredients": [
    { "name": "Veal slice", "type": "Meat","calories": 110 },
    { "name": "Prosciutto crudo", "type": "Meat", "calories": 150},
    { "name": "Sage", "type": "Vegetable", "calories": 50 },
    { "name": "Butter", "type": "Dairy", "calories": 325 }
  ],
  "course": "Main",
  "allergens": ["lactose"]
},
 { "name": "Tiramisu",
  "ingredients": [
    { "name": "Lady fingers", "type": "Cereal","calories": 200 },
    { "name": "Eggs", "type": "Dairy", "calories": 143},
    { "name": "Mascarpone cheese", "type": "Dairy", "calories": 210 },
    { "name": "Coffee", "type": "Vegetable", "calories": 2 },
    { "name": "Cocoa powder", "type": "Vegetable", "calories": 138 }
  ],
```

Figure 2.1: Dish List defined inside the Menu decision

Decide Dishes Without Allergies
dishList

	inputs	outputs	annotations
U	Allergies	Decide Dishes Without Allergies	Description
	allergiesList *"lactose", "eggs", "peanuts", "tree nuts", "fish", "shellfish", "mollusks", "soy", "gluten"*	*dishList*	
1	(for a in (if Allergies instance of list then Allergies else [Allergies]) return a)	Menu[not(some i in item.allergens satisfies i in Allergies)]	

Figure 2.2: Filter Allergies Decision Table

The purpose of this module is to reconcile the individual outputs of the allergenic, dietary profile, and caloric filters so that a coherent set of options is forwarded to the final decision table, named *Suggest Personalized Menu*.

Decide Dishes Match Guest Profile
dishList

	inputs	outputs	annotations
U	Guest Profile	Decide Dishes Match Guest Profile	Description
	dietProfileType *"carnivor", "vegetarian", "vegan", "fish-based"*	*dishList* *nullMenu*	
1	"carnivor"	Menu[some i in item.ingredients satisfies i.type = "Meat"]	
2	"vegetarian"	Menu[every i in item.ingredients satisfies not(i.type = "Meat" or i.type = "Fish")]	
3	"vegan"	Menu[every i in item.ingredients satisfies not(i.type = "Meat" or i.type = "Fish" or i.type = "Dairy")]	
4	"fish-based"	Menu[some i in item.ingredients satisfies i.type = "Fish"]	

Figure 2.3: Filter Guest Profile Decision Table

Decide Dishes Match Calories Profile
dishList

	inputs	outputs	annotations
U	Calories Profile	Decide Dishes Match Calories Profile	Description
	caloriesProfileType *"Light/Diet Friedly [200 - 400 kcal]", "Moderate/Balanced [200 - 700 kcal]", "Hearty/Energy rich [200 - 1000 kcal]", "High - Calorie [1000+ kcal]"*	*dishList* *nullMenu*	
1	"Light/Diet Friedly [200 - 400 kcal]"	Menu[sum(item.ingredients.calories) in [200..400]]	
2	"Moderate/Balanced [200 - 700 kcal]"	Menu[sum(item.ingredients.calories) in [200..700]]	
3	"Hearty/Energy rich [200 - 1000 kcal]"	Menu[sum(item.ingredients.calories) in [200..1000]]	
4	"High - Calorie [1000+ kcal]"	Menu[(sum(item.ingredients.calories) in [200..1000]) or (sum(item.ingredients.calories) > 1000)]	

Figure 2.4: Filter Calories Profile Decision Table

This table is responsible for showing the final suggestion to the user based on the conditions entered previously. A suggestion consists of two pairs of values named *Final Suggestion* and *Correlated Dishes*: the first is used to return a unique menu, while the second returns all the dishes correlated to the user's constraints.

The separation of the two output variables has, as its main goal, the return of a unique menu to the user composed only by a single dish for each course accompanied by a selection of related dishes, in the case the user does not like the dishes recommended by the main choice. Precisely for this reason, a business knowledge model called *Personalized-Menu Model*, was implemented in the final solution. Its task is to obtain the first occurrence of a dish for each course and build a final menu, as illustrated in Figure 2.5.

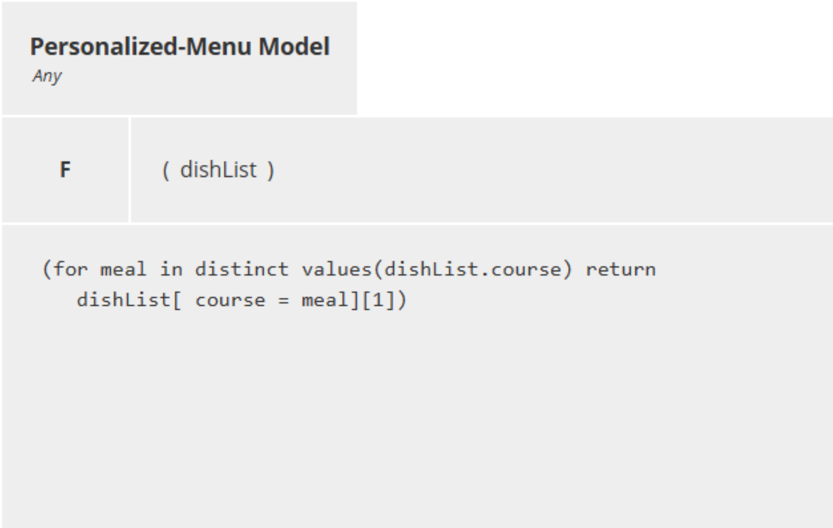


Figure 2.5: Business Knowledge Model adopted

The final decision of *Suggest Personalized Menu* is implemented by two different rules, based on the size of returned filtered list. In particular, if the structure carries at least one element, a final suggestion is returned. Otherwise, a message error is returned for both of the two output variables.

Figure 2.6 presents the final decision table used to provide suggested menu to the user, in particular its structure and how the decision logic was implemented.

This overall approach is encapsulated in the *Decision Model Diagram* presented in Figure 2.7.

2.2 Prolog

In order to complement the decision-table approach, the personalized-menu engine has been reimplemented in Prolog. The following section illustrates how the core domain knowledge is encoded as Prolog facts and how the recommendation logic is expressed through Prolog rules. All guest preferences, ingredient characteristics and dish compositions are first declared as facts; these are then consumed by the inference rules that compute caloric totals, enforce dietary constraints and filter out allergens.

2.2.1 Facts

The knowledge base opens by asserting each diner's declared profile together with any known allergies. The fragment shown in Figure 2.8 demonstrates how a simple pair of predicates, `guest_profile/1` and `allergy/1`, capture the essential information about what a guest eats and what they must avoid.

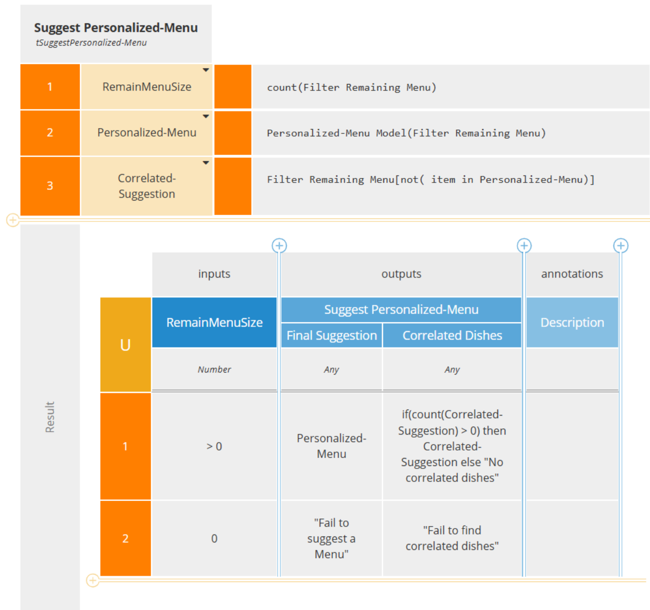


Figure 2.6: Structure of Suggest-Personalized Menu

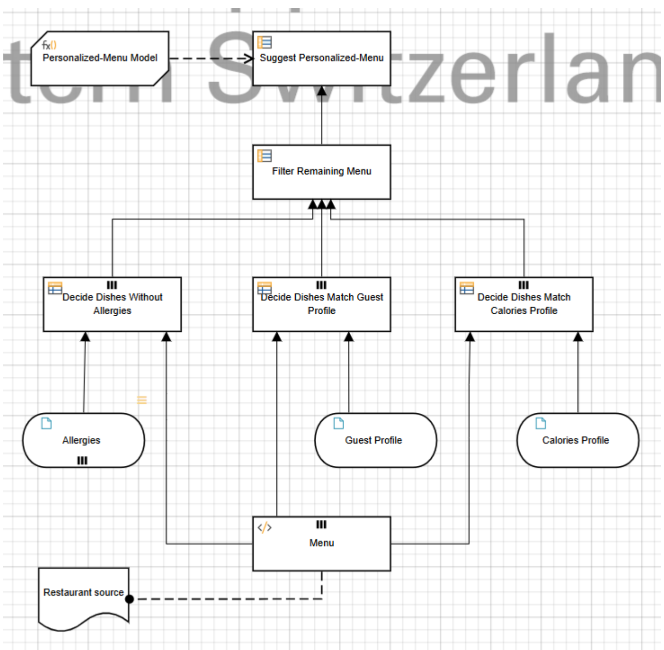


Figure 2.7: Decision Model Diagram

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%           Guest Profiles and Allergies                %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Guest profiles (using atoms without hyphens, e.g., fish_based instead of fish-based)
guest_profile(carnivor).
guest_profile(vegetarian).
guest_profile(vegan).
guest_profile(fish_based).
guest_profile(calorie_conscious).

% Recognized allergies
allergy(lactose).
allergy(gluten).
allergy(soy).
allergy(eggs).
allergy(peanuts).
allergy(tree_nuts).
allergy(shellfish).
allergy(mollusks).
```

Figure 2.8: Prolog facts defining guest profiles and recognized allergens

Following this, ingredient categories and meal courses are declared. As Figure 2.9 shows, `type/1` classifies each ingredient into broad groups such as `meat`, `fish` or `vegetable`, while `course/1` identifies whether a dish is intended as an appetizer, a first course, a main or a dessert.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%             Ingredient Types and Courses              %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Ingredient types
type(vegetable).
type(dairy).
type(meat).
type(fish).
type(cereal).

% Course types
course(appetizer).
course(first).
course(main).
course(dessert).
```

Figure 2.9: Prolog facts for ingredient types and course categories

At the heart of the fact base is a detailed listing of every ingredient and its caloric content. In Figure 2.10 the predicate `ingredient/1` enumerates each item—ranging from `tomato` and `shrimp` to `mascarpone` and `octopus`—while Figure 2.11 shows how `calories_ingredient/2` associates each ingredient with an approximate kilocalorie value.

To link individual ingredients to their categories, the predicate `type_ingredient/2` records, for example, that `shrimp` is of type `fish` and that `eggs` belong to `dairy` (see Figure 2.12). This association is critical both for filtering by diet and for grouping ingredients by allergen.

With all ingredients defined, the set of dishes appears next. Figure 2.13 enumerates

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                  Ingredients and Properties           %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Available ingredients
ingredient(bread).
ingredient(tomato).
ingredient(garlic).
ingredient(basil).
ingredient(rice).
ingredient(shrimp).
ingredient(squid).
ingredient(mussels).
```

Figure 2.10: Prolog listing of ingredients

```
% Calorie counts for ingredients (approximate values)
calories_ingredient(bread, 250).
calories_ingredient(tomato, 18).
calories_ingredient(garlic, 149).
calories_ingredient(basil, 22).
calories_ingredient(rice, 130).
calories_ingredient(shrimp, 85).
calories_ingredient(squid, 92).
calories_ingredient(mussels, 86).
calories_ingredient(parsley, 36).
calories_ingredient(veal, 110).
calories_ingredient(ham, 150).
```

Figure 2.11: Prolog facts for caloric values of ingredients

```
% Association between an ingredient and its type
type_ingredient(bread, cereal).
type_ingredient(tomato, vegetable).
type_ingredient(garlic, vegetable).
type_ingredient(basil, vegetable).
type_ingredient(rice, cereal).
type_ingredient(shrimp, fish).
type_ingredient(squid, fish).
type_ingredient(mussels, fish).
type_ingredient(parsley, vegetable).
type_ingredient(veal, meat).
type_ingredient(ham, meat).
```

Figure 2.12: Prolog associations between ingredients and their types

each menu item via dish/1. The relationship between dishes and ingredients is made explicit by contains_ingredient/2 facts, illustrated in Figure 2.14, which form the basis for computing nutritional totals. Finally, the predicates dish_course/2 and contains_allergy/2, shown in Figures 2.15 and 2.16, respectively bind each dish to its course type and to the allergens it may contain.

```
% Definition of available dishes
dish(tomato_bruschetta).
dish(seafood_risotto).
dish(saltimbocca_alla_romana).
dish(tiramisu).
dish(caprese_salad).
dish(pumpkin_risotto).
dish(grilled_salmon).
dish(panna_cotta).
dish(octopus_salad).
dish(lasagna).
dish(milanese_cutlet).
```

Figure 2.13: Prolog facts enumerating all available dishes

```
% Association of ingredients for each dish
contains_ingredient(tomato_bruschetta, bread).
contains_ingredient(tomato_bruschetta, tomato).
contains_ingredient(tomato_bruschetta, garlic).
contains_ingredient(tomato_bruschetta, basil).

contains_ingredient(seafood_risotto, rice).
contains_ingredient(seafood_risotto, shrimp).
contains_ingredient(seafood_risotto, squid).
contains_ingredient(seafood_risotto, mussels).
contains_ingredient(seafood_risotto, garlic).
contains_ingredient(seafood_risotto, parsley).
```

Figure 2.14: Prolog facts describing which ingredients each dish contains

2.2.2 Rules

Having established the fact base, the recommendation engine is implemented via a small set of recursive and composite rules. The predicate dish_calories/2, depicted in Figure 2.17, gathers every caloric entry for the ingredients of a given dish and sums

```prolog
% Association of dishes with course types
dish_course(tomato_bruschetta, appetizer).
dish_course(seafood_risotto, first).
dish_course(saltimbocca_alla_romana, main).
dish_course(tiramisu, dessert).
dish_course(caprese_salad, appetizer).
dish_course(pumpkin_risotto, first).
dish_course(grilled_salmon, main).
dish_course(panna_cotta, dessert).
dish_course(octopus_salad, appetizer).
dish_course(lasagna, first).
dish_course(milanese_cutlet, main).
```

Figure 2.15: Prolog mapping of dishes to course categories

```prolog
% Association of dishes with possible allergen concerns
contains_allergy(tomato_bruschetta, gluten).

contains_allergy(seafood_risotto, shellfish).
contains_allergy(seafood_risotto, mollusks).

contains_allergy(saltimbocca_alla_romana, lactose).

contains_allergy(tiramisu, lactose).
contains_allergy(tiramisu, eggs).
contains_allergy(tiramisu, gluten).

contains_allergy(caprese_salad, lactose).

contains_allergy(pumpkin_risotto, lactose).

contains_allergy(panna_cotta, lactose).

contains_allergy(octopus_salad, mollusks).

contains_allergy(lasagna, gluten).
contains_allergy(lasagna, lactose).

contains_allergy(milanese_cutlet, gluten).
contains_allergy(milanese_cutlet, eggs).
```

Figure 2.16: Prolog annotation of dish-specific allergens

them to yield the dish's total energy content. This computed value drives subsequent diet-based checks.

```prolog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%            Calculation of Dish Calories             %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Calculate the total calories of a dish by summing the calories of its ingredients.
dish_calories(Dish, TotalCalories) :-
    findall(Cal, (contains_ingredient(Dish, Ingredient), calories_ingredient(Ingredient, Cal)), CaloriesList),
    sum_list(CaloriesList, TotalCalories).
```

Figure 2.17: Prolog rule calculating total calories per dish

The central logic for generating recommendations resides in the predicate `recommended_dish/3`, illustrated in Figure 2.18. This predicate succeeds when a candidate dish satisfies every declared profile constraint, avoids all specified allergens, and naturally falls within any caloric boundaries implied by the profiles. Internally, it invokes the helper predicates `check_profiles/2` and `check_allergies/2`.

```prolog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%          Rules for Dish Recommendation            %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Main predicate: recommended_dish(Profiles, Allergies, Dish)
% Profiles is a list of profiles that must all be satisfied.
% Allergies is a list of allergens to avoid.
recommended_dish(Profiles, Allergies, Dish) :-
    dish(Dish),
    check_profiles(Profiles, Dish),
    check_allergies(Allergies, Dish).
```

Figure 2.18: Prolog main predicate driving the recommendation system

Profile verification is performed by the composite predicate `check_profiles/2`, which recursively ensures that each profile in the user's list is satisfied by the dish. The individual profile checks, examples of which appear in Figures 2.19 and 2.20, enforce rules such as "no meat for vegetarians," "no dairy or meat for vegans," or a minimum presence of fish for a fish-based diet.

Allergy avoidance is handled by `check_allergies/2`, which succeeds only if none of the undesired allergens is associated with the dish. Through the harmonious interplay of these rules and the foundational facts, the Prolog engine is capable of producing a personalized menu that respects every guest's nutritional goals and health restrictions.

Finally, we created some predefined guest profiles for showing suggested menu during queries thanks to the predicate `suggest_menu/2` as in Figure 2.21.

2.3 Knowledge graph/Ontology

The definitive ontology for Task 3 is an OWL 2 DL artefact. All entities declared in the file follow that namespace, while the usual `rdf`, `rdfs`, `owl` and `xsd` prefixes are retained for meta-modelling. The file is encoded in Turtle and was authored directly in Protégé, then refined with the OWL API during automated regression tests. The model mirrors the domain already captured by the DMN decision tables and the Prolog

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                Composite Profile Checks               %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% check_profiles(Profiles, Dish) succeeds if Dish satisfies every profile in the Profiles list.
check_profiles([], _).
check_profiles([Profile|Rest], Dish) :-
    check_profile(Profile, Dish),
    check_profiles(Rest, Dish).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                Profile-based Checks                   %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% For a vegetarian: no ingredient of type meat must be present.
check_profile(vegetarian, Dish) :-
    \+ ( contains_ingredient(Dish, Ingredient),
         type_ingredient(Ingredient, meat)
       ).

% For a vegan: no ingredient of type meat or dairy must be present.
check_profile(vegan, Dish) :-
    \+ ( contains_ingredient(Dish, Ingredient),
         ( type_ingredient(Ingredient, meat) ; type_ingredient(Ingredient, dairy) )
       ).

% For a carnivor: at least one ingredient of type meat should be present.
check_profile(carnivor, Dish) :-
    contains_ingredient(Dish, Ingredient),
    type_ingredient(Ingredient, meat).

% For a fish_based guest: at least one ingredient of type fish should be present.
check_profile(fish_based, Dish) :-
    contains_ingredient(Dish, Ingredient),
    type_ingredient(Ingredient, fish).
```

Figure 2.19: Prolog helper rules for the first set of profile checks

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%          Calorie-based Diet Category Checks           %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% For a light (diet friendly) profile: the dish must be between 200 and 400 kcal.
check_profile(light, Dish) :-
    dish_calories(Dish, Total),
    Total >= 200,
    Total =< 400.

% For a moderate (balanced) profile: the dish must be between 401 and 700 kcal.
check_profile(moderate, Dish) :-
    dish_calories(Dish, Total),
    Total > 400,
    Total =< 700.

% For a hearty (energy rich) profile: the dish must be between 701 and 1000 kcal.
check_profile(hearty, Dish) :-
    dish_calories(Dish, Total),
    Total > 700,
    Total =< 1000.

% For a high calorie profile: the dish must be above 1000 kcal.
check_profile(high, Dish) :-
    dish_calories(Dish, Total),
    Total > 1000.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%            Allergy-based Checks                       %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% check_allergies(Allergies, Dish) succeeds if none of the allergens in Allergies is present in the Dish.
check_allergies([], _).
check_allergies([Alg | Rest], Dish) :-
    \+ contains_allergy(Dish, Alg),
    check_allergies(Rest, Dish).
```

Figure 2.20: Prolog helper rules for the second set of profile checks

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%            Guests-Profile Registered                %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% suggest_menu(NameGuest, Result) return a menu suggestion based on the user's preferences. Please, consider that
% all the following queries rappresent a pre-built profiles with a set of fixed preferences. If you would obtain a
% custom suggestion it's necessary to execute recommended_dish() query.

suggest_menu("Samuele", Suggest) :- recommended_dish(_, light, _, Suggest).
suggest_menu("Matteo", Suggest) :- recommended_dish(vegetarian, hearty, [eggs, gluten], Suggest).
suggest_menu("Francesco", Suggest) :- recommended_dish(carnivor, high, _, Suggest).
suggest_menu("Enrico", Suggest) :- recommended_dish(vegan, moderate, [lactose], Suggest).
```

Figure 2.21: Suggest Menu Prolog profiles predefined

facts of as in Sections 2.1 and 2.2, yet it enriches it with formal semantics, reasoning support and an explicit rule layer.

2.3.1 Protégé

In Protégé the first step consisted in declaring the core classes `Dish`, `Ingredient`, `Course`, `Allergy`, `CaloriesDesired`, `FoodProfile` and `Guest`. Nutritional categories (`Meat`, `Fish`, `Dairy`, `Cereal`, `Vegetable`) are modelled as disjoint subclasses of `Ingredient`, an axiom made explicit by an `owl:AllDisjointClasses` construct so that the reasoner will catch mis-classifications (see Figure2.22). Composite dish types such as `MeatDish`, `FishDish`, `VegetarianDish` and `VeganDish` are defined through qualified cardinality restrictions on the object property `containsIngredient`; for example, `MeatDish` is declared equivalent to "some `containsIngredient Meat`". Object-property modelling follows a systematic inverse pattern: `containsIngredient` is paired with `isContainedBy`, `contains_allergy` with `allergyIsContained`, `belongs_to_the` with `course_includes`, `hasPersonalizedMenuDish` with `inPersonalizedMenu`
Domains and ranges are asserted for every property so that OWL validation can reveal illegal statements at authoring time (see Figure **??**).

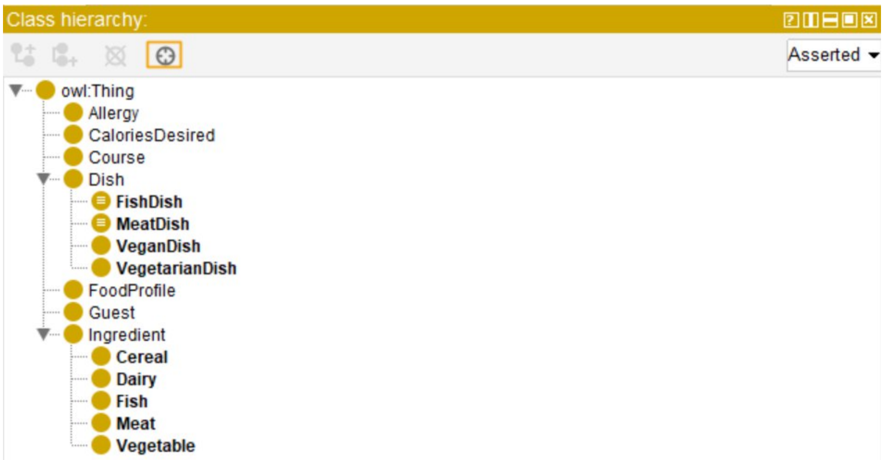


Figure 2.22: Entities in Protegé

Quantitative information is attached through datatype properties. Each `Ingredient` carries a mandatory `hasCalories` assertion, while `Dish` instances expose a precomputed `hasTotalCalories` value. Desired calorie bands requested in the specification (*Light*, *Moderate*, *Hearty*, *High*) are represented as individuals of `CaloriesDesired` equipped with `hasMinCalories` and `hasMaxCalories` literals; the open-world assumption is therefore tamed by closing the numeric interval explicitly (see Figure **??**).

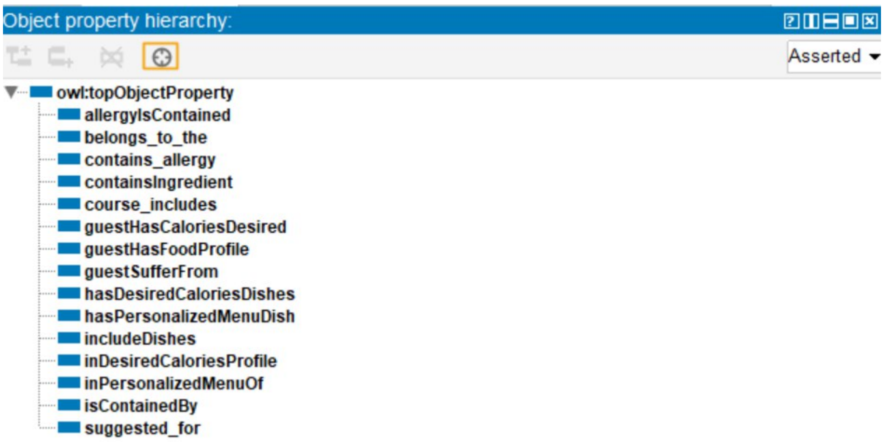


Figure 2.23: Object Properties ins Protegé

The modelling session concluded by instantiating the full menu. Each of the eleven dishes, their forty ingredients, the four courses and the five guest profiles were entered as named individuals; cross-links among them comply with the domains and ranges previously fixed and with the inverses automatically maintained by HermiT. After classification the reasoner placed, for instance, `Grilled_Salmon` under both `Dish` and the inferred subclass `FishDish`, proving that the axioms were coherent.

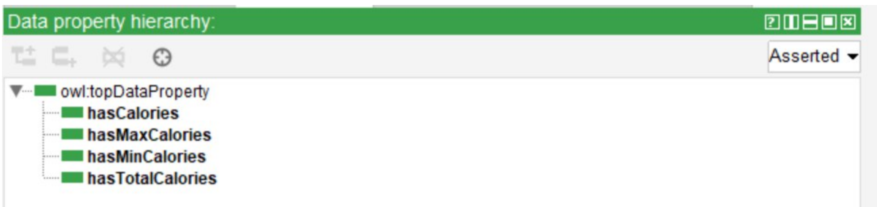


Figure 2.24: Data Propertied in Protégé

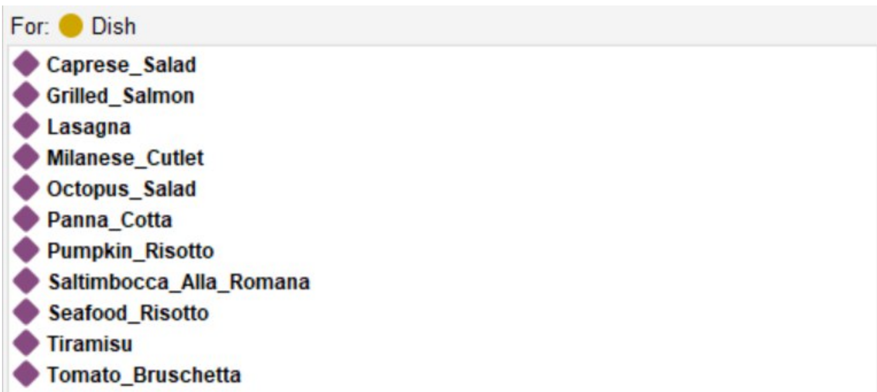
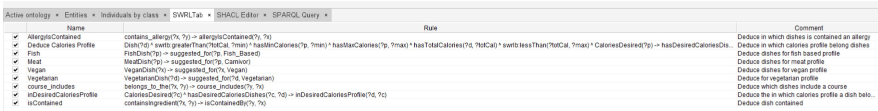


Figure 2.25: Individuals for Dish Entity on Protegé

2.3.2 SWRL

The ontology relies on a compact yet expressive rule layer written in SWRL and activated through the SWRLTab (see Figure 2.26). The annotation property `swrla:isRuleEnabled` is used to toggle individual rules for debugging; all rules shipped with the release are flagged as `true`. One group of rules propagates inverse relations that would otherwise require explicit duplication. The rule labelled *isContained* states that if a dish `x` `containsIngredient` `y`, then `y` `isContainedBy` `x`. A symmetrical pattern establishes `course_includes` from `belongs_to_the`, and `allergyIsContained` from `contains_allergy`. These rules merely materialise data that a SPARQL client can exploit without relying on reasoning at query time.



Active ontology × Entities × Individuals by class × SWRLTab × SHACL Editor × SPARQL Query ×

	Name	Rule	Comment
✔	AllergyIsContained	contains_allergy(?x, ?y) -> allergyIsContained(?y, ?x)	Deduce in which dishes is contained an allergy
✔	Deduce Calories Profile	Dish(?d) ^ swrlb:greaterThan(?totCal, ?min) ^ hasMinCalories(?p, ?min) ^ hasMaxCalories(?p, ?max) ^ hasTotalCalories(?d, ?totCal) ^ swrlb:lessThan(?totCal, ?max) ^ CaloriesDesired(?p) -> hasDesiredCaloriesDis...	Deduce in which calories profile belong dishes
✔	Fish	FishDish(?p) -> suggested_for(?p, Fish_Based)	Deduce dishes for fish based profile
✔	Meat	MeatDish(?p) -> suggested_for(?p, Carnivor)	Deduce dishes for meat profile
✔	Vegan	VeganDish(?x) -> suggested_for(?x, Vegan)	Deduce dishes for vegan profile
✔	Vegetarian	VegetarianDish(?d) -> suggested_for(?d, Vegetarian)	Deduce for vegetarian profile
✔	course_includes	belongs_to_the(?x, ?y) -> course_includes(?y, ?x)	Deduce which dishes include a course
✔	inDesiredCaloriesProfile	CaloriesDesired(?c) ^ hasDesiredCaloriesDishes(?c, ?d) -> inDesiredCaloriesProfile(?d, ?c)	Deduce the in which calories profile a dish belo...
✔	isContained	containsIngredient(?x, ?y) -> isContainedBy(?y, ?x)	Deduce dish contained

Figure 2.26: SWRL Rules

A second group assigns dishes to qualitative menu profiles. Three single-premise rules map any instance of `MeatDish`, `FishDish` or `VeganDish` to the appropriate `FoodProfile` via the object property `suggested_for`. Two further rules with identical heads are provided so that both variable-bound and constant-bound formulations can be toggled while measuring performance. A third group calculates diet suitability based on total calories. The rule labelled *Deduce Calories Profile* takes a dish `d` together with its `hasTotalCalories` value, compares that literal against the `hasMinCalories` and `hasMaxCalories` of a calories band `p` through built-in arithmetic atoms, and if the interval comparison succeeds it asserts `hasDesiredCaloriesDishes(p, d)`; the companion rule then infers `inDesiredCaloriesProfile(d, p)` to allow OWL class expressions to refer to such dishes directly. All rules have been sanity-checked under the Hybrid reasoner and produce exactly the triples expected from the manual sample calculations in Tasks in Sections 2.1 and 2.2.

The ontology is therefore ready for deployment in any triple store that supports SWRL materialisation, SHACL validation and SPARQL 1.1 querying. It exposes richer semantics than the previous artefacts while remaining completely aligned with them, thereby ensuring that downstream BPMN processes can retrieve personalised menus by a single parameterised SPARQL query, free from procedural glue code.

2.3.3 SPARQL Queries

The ontology is consumed through a small set of parameterised SPARQL 1.1 queries that transform the inferred triples into the artefacts needed by the personalised–menu application. Each query is designed for direct execution in a Fuseki endpoint; names and IRIs are written in the same namespace used by the OWL file so that no additional prefixes are required. Where run-time parameters occur, they are supplied via `VALUES` clauses rather than string concatenation, a strategy that keeps the query text itself static and cache-friendly while still allowing simple injection from the BPMN engine. All filters that might discard a solution are expressed through `FILTER NOT EXISTS` patterns, thereby preserving monotonicity under entailment. The queries shipped with Task 3 are reported verbatim below.

Query 1 – Retrieve vegetarian, moderate-calorie dishes that avoid a given allergy. The first query returns the triple `(?dish, ?course, ?calories)` for every dish classified as `Vegetarian`, belonging to the calorie band `Moderate` and not containing the allergy passed in the `?excludedAllergy` variable. Two `BIND` expressions translate the matching of the food profile and the calorie profile into Boolean flags; their product (`?suggestion`) is then compared against 0, ensuring that a row survives the filter only when both aspects are satisfied.

```
PREFIX : <http://www.semanticweb.org/kebi.task3.pirani-machella#>

SELECT ?dish ?course ?calories
WHERE {
  VALUES ?desiredFoodProfile   { :Vegetarian }
  VALUES ?desiredCalorieProfile { :Moderate }
  VALUES ?excludedAllergy      { :Shellfish }

  ?dish a :Dish ;
        :hasTotalCalories ?calories ;
        :belongs_to_the ?course ;
        :suggested_for ?foodprofile ;
        :inDesiredCaloriesProfile ?caloriesprofile .

  BIND(IF(?foodprofile     = ?desiredFoodProfile,   1, 0) AS ?v1)
  BIND(IF(?caloriesprofile = ?desiredCalorieProfile, 1, 0) AS ?v2)
  BIND((?v1 * ?v2) AS ?suggestion)

  FILTER(?suggestion > 0)
  FILTER NOT EXISTS { ?dish :contains_allergy ?excludedAllergy }
}
```

Query 2 – Construct a personalised menu for every guest present in the graph. The second query is formulated in the `CONSTRUCT` form so that its result can be inserted back into the triple store as materialised links between guests and suitable dishes. It iterates simultaneously over all `:Guest` and `:Dish` individuals, filters away dishes that violate declared allergies, and checks whether both the food profile and the calorie profile required by the guest coincide with those of the dish. Missing requirements are tolerated thanks to `OPTIONAL` and `FILTER` conditions that treat unbound variables as "no constraint".

```
PREFIX : <http://www.semanticweb.org/kebi.task3.pirani-machella#>

CONSTRUCT {
  ?guest :hasPersonalizedMenuDish ?dish .
}
WHERE {
  ?guest a :Guest .
  ?dish  a :Dish .

  OPTIONAL { ?guest :guestHasFoodProfile     ?fp  }
  OPTIONAL { ?dish  :suggested_for          ?fp_d }

  OPTIONAL { ?guest :guestHasCaloriesDesired ?cp  }
  OPTIONAL { ?dish  :inDesiredCaloriesProfile ?cp_d }
```

```
  FILTER NOT EXISTS { ?guest :guestSufferFrom ?a .
                      ?dish  :contains_allergy ?a . }

  FILTER (!BOUND(?fp) || ?fp = ?fp_d)
  FILTER (!BOUND(?cp) || ?cp = ?cp_d)
}
```

Query 3 – Construct a personalised menu for a specific guest. When the calling application already knows which guest is active, the previous query can be specialised by binding the `?guest` variable to a constant—`:Samuele` in the example—through a `BIND` expression. The remainder of the query is identical, guaranteeing consistent semantics across both use cases. This version is particularly convenient for embedding directly into a BPMN task where the guest identifier is available as a process variable.

```
PREFIX : <http://www.semanticweb.org/kebi.task3.pirani-machella#>

CONSTRUCT {
  ?guest :hasPersonalizedMenuDish ?dish .
}
WHERE {
  BIND(:Samuele AS ?guest)
  ?dish a :Dish .

  OPTIONAL { ?guest :guestHasFoodProfile     ?fp  }
  OPTIONAL { ?dish  :suggested_for          ?fp_d }

  OPTIONAL { ?guest :guestHasCaloriesDesired ?cp  }
  OPTIONAL { ?dish  :inDesiredCaloriesProfile ?cp_d }

  FILTER NOT EXISTS { ?guest :guestSufferFrom ?a .
                      ?dish  :contains_allergy ?a . }

  FILTER (!BOUND(?fp) || ?fp = ?fp_d)
  FILTER (!BOUND(?cp) || ?cp = ?cp_d)
}
```

The three queries together complete the knowledge-based stack: SWRL materialises the implicit `suggested_for` and `inDesiredCaloriesProfile` links, while SPARQL uses that information to assemble a guest-specific menu, ready to be streamed into the mobile front-end or injected into the BPMN execution context without any additional transformation.

2.3.4 SHACL Shapes

To guarantee that every RDF graph uploaded to the triplestore respects the structural and numerical constraints that underpin menu reasoning, a dedicated SHACL shape graph has been authored in the same namespace as the ontology. The validation layer is designed to be both strict enough to detect modelling mistakes early and permissive enough to support incremental enrichment of the dataset. Each domain class that can be instantiated by end-users—`Dish`, `Ingredient`, `Course`, `FoodProfile`, `CaloriesDesired` and `Guest`—is associated with a `sh:NodeShape`. The shapes use closed-world cardinality checks for mandatory links, datatype restrictions for numeric literals and range guards such as `sh:minInclusive` 0 to rule out negative calorie values before they reach the SWRL layer. Optional relationships are bound by `sh:maxCount` rather than hard `sh:minCount` so that their absence signals "no preference" rather than an error, thereby aligning SHACL validation with the semantics of the SPARQL queries that treat unbound variables as wildcards.

The following Turtle fragment shows the complete shape graph shipped; it can be loaded into any SHACL-aware engine—Fuseki, GraphDB or PySHACL—for batch validation or plugged into real-time data-entry pipelines to provide immediate feedback.

```
@prefix :      <http://www.semanticweb.org/kebi.task3.pirani-machella#> .
@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#> .

:DishShape a sh:NodeShape ;
  sh:targetClass :Dish ;
  sh:property [
      sh:path :belongs_to_the ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
  ] ;
  sh:property [
      sh:path :containsIngredient ;
      sh:minCount 1 ;
  ] ;
  sh:property [
      sh:path :suggested_for ;
      sh:minCount 1 ;
  ] ;
  sh:property [
      sh:path :inDesiredCaloriesProfile ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
  ] ;
  sh:property [
      sh:path :hasTotalCalories ;
      sh:datatype xsd:integer ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:minInclusive 0 ;
  ] .

:IngredientShape a sh:NodeShape ;
```

```
  sh:targetClass :Ingredient ;
  sh:property [
      sh:path :hasCalories ;
      sh:datatype xsd:integer ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:minInclusive 0 ;
  ] ;
  sh:property [
      sh:path :isContainedBy ;
      sh:minCount 1 ;
  ] .

:CourseShape a sh:NodeShape ;
  sh:targetClass :Course ;
  sh:property [
      sh:path :course_includes ;
      sh:minCount 1 ;
  ] .

:FoodProfileShape a sh:NodeShape ;
  sh:targetClass :FoodProfile ;
  sh:property [
      sh:path :includeDishes ;
      sh:minCount 1 ;
  ] .

:CaloriesDesiredShape a sh:NodeShape ;
  sh:targetClass :CaloriesDesired ;
  sh:property [
      sh:path :hasDesiredCaloriesDishes ;
      sh:minCount 1 ;
  ] ;
  sh:property [
      sh:path :hasMinCalories ;
      sh:datatype xsd:integer ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:minInclusive 0 ;
  ] ;
  sh:property [
      sh:path :hasMaxCalories ;
      sh:datatype xsd:integer ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:minInclusive 0 ;
  ] .

:GuestShape a sh:NodeShape ;
```

```
sh:targetClass :Guest ;
sh:property [
    sh:path :guestHasFoodProfile ;
    sh:maxCount 1 ;
] ;
sh:property [
    sh:path :guestHasCaloriesDesired ;
    sh:maxCount 1 ;
] .
```

Together with the OWL axioms and SWRL rules, these SHACL shapes complete the three-layer knowledge representation stack: OWL for closed-world conceptual integrity, SWRL for deductive enrichment and SHACL for runtime data quality assurance. The result is a robust, self-validating knowledge graph that can safely drive personalised menu recommendations in production.

2.4 Summary

This chapter has outlined the progressive refinement of the personalised-menu engine from rule-based artefacts to a fully fledged knowledge graph. Decision Tables were introduced first to partition the restaurant's dish list along three orthogonal dimensions—allergenic safety, dietary suitability and caloric adequacy—so that each constraint could be maintained, tested and evolved independently. The same logic was then re-expressed in Prolog, where explicit facts captured every ingredient, dish, guest preference and allergen, while recursive rules computed calorie totals and enforced the composite filtering conditions required for a valid recommendation.

Building on this foundation, the solution migrated to an OWL 2 DL ontology that preserves functional equivalence with the earlier artefacts yet enriches them with formal semantics. Protégé was used to declare the core classes, disjoint ingredient categories, inverse object properties and quantitative data properties; SWRL rules materialise inverse relations, infer diet profiles and assign dishes to calorie bands without duplicating data; SPARQL 1.1 queries transform the inferred triples into guest-specific menus ready for downstream BPMN tasks; and a dedicated SHACL shape graph enforces structural integrity and numeric bounds at ingestion time. Collectively, these layers provide a coherent and self-validating knowledge-based solution in which declarative constraints, deductive enrichment and query-time retrieval cooperate seamlessly to deliver accurate, explainable and maintainable recommendations.

3. Agile and ontology-based meta-modeling

In our endeavour to enrich classical business process notation with the semantic precision of ontologies, we have adopted an agile meta-modelling methodology that seamlessly fuses BPMN 2.0 constructs with an OWL knowledge graph and accompanying SWRL rules and SPARQL queries. This approach acknowledges that the domain of personalized menu recommendation is inherently dynamic: new dietary profiles emerge, ingredients change with seasonality, and allergen information must be maintained with utmost accuracy.

The foundation of our solution is the Personalized Menu Ontology, authored in Turtle syntax and encompassing classes such as `:Dish`, `:Ingredient`, `:Course`, and `:Profile`. Nutritional attributes, including caloric values, are modelled as data properties, while preferences and restrictions—ranging from vegetarian inclinations to lactose and gluten intolerances—are captured as object properties linking dishes to profile instances. Upon loading the ontology into Apache Jena Fuseki, accessible at `http://localhost:3030`, we exercise the expressiveness of SPARQL to retrieve candidate dishes that satisfy a guest's declared `lo:FoodProfile` and `lo:CaloriesProfile`, while filtering out any instances that contravene specified allergies. The SPARQL endpoint thus becomes not merely a repository but an active reasoning component, executing our parameterized query and returning, for instance, *Caprese_Salad*, *Tomato_Bruschetta*, and *Tiramisu* along with their caloric totals, as depicted in Figure 3.1.

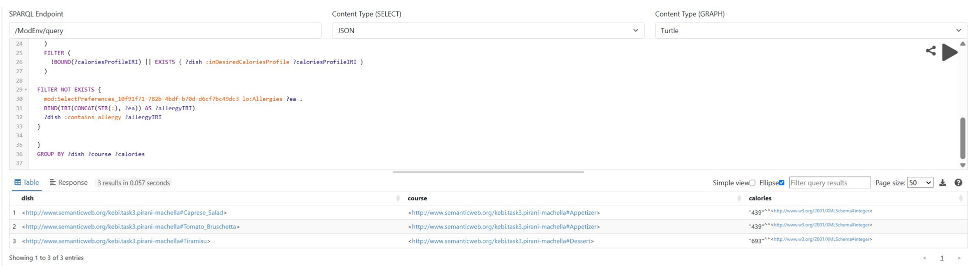


Figure 3.1: SPARQL query results in Jena Fuseki (localhost:3030).

Building upon this semantic substrate, the Agile and Ontology-Aided Meta-Modelling Environment (AOAME), deployed at `http://localhost:4200`, presents an extended BPMN palette in which the generic `Task` element is specialized into `:SuggestMealsTask`. This new modelling element is underpinned by two bespoke properties, `:hasQuery` and `:endpointURL`, which respectively encapsulate the SPARQL SELECT statement and the URI of the Fuseki service. Graphically, the `Guest` pool has different color of the rectangle depicting tasks, visually differentiating it from routine tasks and signalling to the restaurant manager that this step invokes semantic reasoning. Such visual cues, combined with the underlying meta-model, streamline the process of constructing and understanding personalized recommendation workflows (see Figure 3.2).

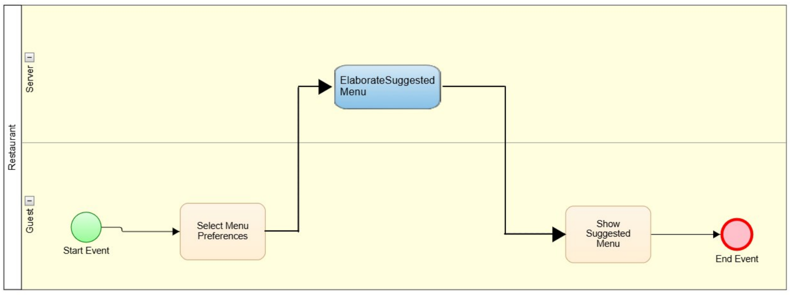


Figure 3.2: Extended BPMN diagram in AOAME (localhost:4200) featuring `SuggestMealsTask`.

A salient feature of our implementation is the ability to parameterize process elements through model-level attributes. As illustrated in Figure 3.3, the user may specify values for `FoodProfile`, `CaloriesProfile`, and `Allergies` directly within the properties inspector of tasks such as `ElaborateSuggestedMenu`. AOAME then propagates these values into the corresponding RDF triples, thereby updating the Fuseki dataset in real time. This bidirectional binding ensures that the process model remains the authoritative source of truth for both control flow and data semantics.

At runtime, when a guest commences the interaction by invoking the "Select Menu Preferences" task, AOAME translates the selected options into SWRL rule inputs and SPARQL query parameters. Upon execution of the specialized `SuggestMealsTask`, an HTTP GET request is issued against Fuseki's SPARQL endpoint. The JSON-formatted response is then parsed by the AOAME engine, which instantiates BPMN data objects representing `:Dish` individuals, annotated with course and caloric data. These data objects are seamlessly bound to the subsequent "Show Suggested Menu" activity, enabling the user interface to render a tailored list of dishes.

From an agile perspective, this architecture decouples the semantic backend from the process notation, allowing ontology engineers to refine class hierarchies, add new dietary profiles, or introduce additional reasoning rules without necessitating changes to the BPMN runtime. Conversely, business analysts can adapt the process flow—adding approval gateways, logging tasks, or alternative recommendation strategies—while preserving the integrity of the underlying knowledge graph. This symbiotic relationship fosters rapid iteration and continuous delivery, both of which are hallmarks of agile development.

In conclusion, the integration of BPMN 2.0 with ontology-based knowledge representation and SPARQL querying yields a robust, extensible framework for personalized menu recommendation. The meta-modelling approach ensures that domain experts and process designers operate in concert, leveraging familiar notations enriched by formal semantics. As a result, restaurant managers can define sophisticated recommendation workflows with minimal technical overhead, while guests enjoy a truly tailored dining experience.

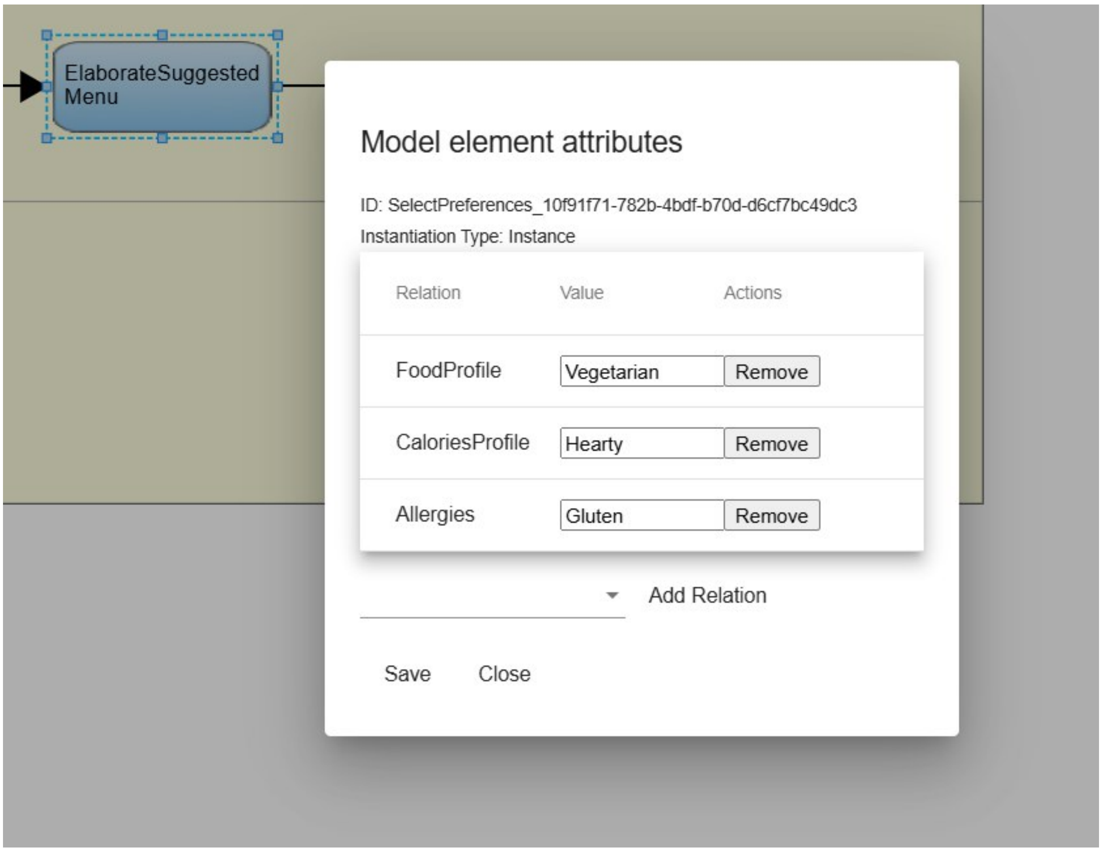


Figure 3.3: Model element attributes in AOAME, linking process parameters to ontology individuals.

4. Conclusions

In this final chapter, we draw together the insights gained throughout the project, presenting first Matteo Machella's reflections (Section 4.1) and then Samuele Pirani's independent viewpoint (Section 4.2).

4.1 Matteo Machella's Considerations

When I look back on our exploration of knowledge-based formalisms, I am struck by how decision tables initially impressed me as a *crystal-clear* medium for capturing business rules. Their grid-like layout mirrors the mental maps of domain experts, and I could readily imagine non-technical stakeholders validating every row and column before we even touched code. Yet as soon as the decision logic acquired a modicum of complexity—with interwoven conditions or multi-dimensional preferences—the simplicity evaporated. Manually crafting branches without resorting to FEEL syntax soon felt like pushing a boulder uphill: the moment we introduced FEEL expressions to tame complexity, our decision tables ceased to be the lightweight, stakeholder-friendly artifact we prized and began to resemble a hidden script for a rules engine.

In sharp contrast, Prolog revealed itself as a **delightful** ally. Its very design encourages you to think in terms of facts and relations, and once the initial predicate schema was in place, extending the system was almost playful. The automated backtracking spared me from writing tedious loops or guards, and the code maintained a *clean*, logical structure even as requirements evolved. For me, this was the clearest demonstration that a logic-programming paradigm can both simplify development and enhance maintainability, especially when rules must adapt over time.

Turning to ontology engineering with Protegé, I embraced the theoretical elegance of OWL and appreciated the rigor of SHACL constraints in preserving data integrity. Crafting classes, properties and individuals felt intellectually satisfying, and invoking SPARQL queries against our triple store underscored how a knowledge graph can serve as a living, queryable backbone. Nonetheless, the user interfaces of Protegé struck me as somewhat austere. Hours spent navigating nested panels, wrestling with prefix declarations and deciphering cryptic console messages dulled the excitement of semantic modelling.

Finally, the AOAME environment promised an agile fusion of BPMN and semantic reasoning. Conceptually, embedding a `SuggestMealsTask` wired to SPARQL felt like a breakthrough: the process model could drive live queries, and vice versa. In practice, however, the clumsy property-inspector and sluggish UX reminded me that *usability* is as crucial as theoretical power. Although I am convinced of AOAME's potential to transform process-centric applications, I remain cautious until the platform's interface matures to match its semantic sophistication.

4.2 Samuele Pirani's Considerations

From my vantage point, decision tables proved to be an excellent pedagogical tool for articulating straightforward decision logic. Their spreadsheet-style representation fostered immediate clarity: if you can see a condition in a cell, you can trust that stakeholders grasp its effect. Yet the limitations of our chosen tool surfaced quickly. Without built-in facilities for testing or versioning, expanding the table to cover nuanced cases became *tedious*, and the lack of modular composition threatened to turn our neat matrix into an unruly labyrinth.

By contrast, Prolog stood out as the **undisputed highlight** of our endeavours. The moment I began defining dishes, ingredients and user profiles as Prolog facts, the language's backtracking mechanism felt like magic. Adding a new dietary rule required only a few lines of code, and I witnessed the inference engine orchestrate the search for valid menu items with astonishing efficiency. In my opinion, this task exemplified how logic programming can streamline the development of adaptive, knowledge-driven systems.

When I shifted to Protegé for ontology development, I appreciated how effortlessly one could create classes and shape graphs. The visual canvas is intuitive, and SHACL validations provided a robust safety net against modelling errors. Nonetheless, I found SWRL's expressiveness wanting: its rule syntax lacks the flexibility to express more sophisticated inference patterns, and debugging complex SWRL rules became a stumbling block rather than a bridge to deeper semantics.

Lastly, AOAME's vision of semantic meta-modelling in BPMN resonated with me in theory: binding process elements directly to SPARQL queries and ontology individuals suggested a seamless design-to-execution pipeline. Yet using the tool in our lab felt akin to navigating a maze with dim lighting. The interface was *unforgiving*, error messages sparse and ambiguous, and simple tasks often required arcane workarounds. Until the platform's user experience is substantially refined, I remain unconvinced of its readiness for everyday enterprise use.