



Personalized Menu Project

MSc Computer Science, Knowledge Engineering course

Fabio Grelloni Mat.132549

Claudio Cozzolino Mat.132015

A.Y. 2024/25

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

Contents

1	Introduction	6
2	Project Objective	6
3	Menu Structure and Testing Dataset	7
4	Decision Model: Trisotech	8
4.1	Decision Modeler	8
4.2	Elements of the Decision Diagram	9
4.2.1	Input Data	9
4.2.2	Error Handling and Robustness	10
4.3	Decision Blocks in the DMN Model	10
4.3.1	Functionality of the Decision Blocks	10
4.3.2	Main Block: Recommend Meals	11
4.3.3	Testing and Validation	11
5	SWISH Prolog	12
5.1	Prolog in the Personalized Menu Project	12
5.2	Facts Definition in Prolog	13
5.2.1	Meal Declaration	13
5.2.2	Meal Categories	13
5.2.3	Calorie Information	13
5.2.4	Dietary Tags and Allergen Indicators	14
6	Prolog Rules: Meal Compatibility and Recommendation Logic	14
6.1	Diet Compatibility Rules	15
6.2	Allergen Checks	15
6.3	Calorie-Conscious Meal Filter	16
6.4	Overall Meal Compatibility Checks	16
6.4.1	meal_safe_for_allergies/2	16
6.4.2	compatible_meal/4	16
6.4.3	compatible_meals/4	16
6.5	Meal Compatibility with Meal Type	16
6.5.1	compatible_meal_with_type/5	17
6.5.2	compatible_meals_with_type/4	17
6.6	Ontology Engineering Role	17
7	Knowledge Representation Technologies	18
7.1	Turtle Syntax	18
7.2	Protégé	18
7.3	Our Ontology	18
7.4	Classes	18
7.4.1	Meal Class	19
7.4.2	Guest Class	20
7.4.3	Ingredients Class	21
7.4.4	DietType Class	21
7.4.5	Categories Class	21
7.4.6	CaloriePreference Class	22
7.4.7	Allergen Class	22
7.5	SWRL Rules	22

7.5.1	Built-in Atom Limitation in Hermit	23
7.5.2	1. Allergy Check	23
7.5.3	2. Calorie-Based Recommendation	24
7.5.4	3. Heavy Meal Definition	24
7.5.5	4. Light Meal Definition	24
7.5.6	5. Vegetarian Diet Recommendation	25
7.6	SHACL Shapes	25
7.6.1	AnimalIngredientShape	25
7.6.2	MealIngredientShape	25
7.6.3	MealShape	26
7.6.4	GuestNameShape	26
7.6.5	GuestShape	26
7.6.6	SHACL Validation Process	26
7.7	SPARQL Queries	27
7.7.1	1. Allergy-based Meal Filtering	27
7.7.2	2. Calorie-Conscious Guests – High-Calorie Meal Filtering	28
7.7.3	3. Vegetarian Guests – Animal-Based Ingredient Exclusion	28
7.7.4	4. Recommended Meals for a Specific Guest	28
7.7.5	5. Explain Why a Meal is Not Recommended	28
7.7.6	6. List All Meals and Their Caloric Values	29
7.7.7	7. Guest Profiles – Dietary and Calorie Preferences	29
7.7.8	8. Meals with Allergens and Their Ingredients	29
8	Ontology-Aware Process Modeling	30
8.1	Agile, Ontology-Based Metamodeling (AOAME)	30
8.2	Business Process Modeling with BPMN 2.0	30
8.2.1	What is BPMN 2.0	30
8.2.2	Our BPMN Model	30
8.3	Jena Fuseki	32
8.3.1	Query in Jena Fuseki	32
9	Conclusion Claudio	36
9.1	Advantages and Disadvantages of the Knowledge-Based Solutions	36
9.1.1	Decision Tables with Trisotech Decision Modeler	36
9.1.2	Prolog Logic Programming	36
9.1.3	Ontology and Knowledge Graph with Jena Fuseki	36
10	Conclusion Fabio	37
10.1	Semantic Ontology Engineering: From Modeling to Inference	37
10.2	Integrating Semantic Models with Business Processes	37
10.3	A Comparative Perspective: Toward a More Elegant Semantic Architecture	38
11	Final Considerations	38

Listings

1	Meal declaration	13
2	Example of meal declaration	13
3	Meal categories	13
4	Example of categorized meals	13
5	Calorie structure	13
6	Example of calories per meal	13
7	Dietary tags	14
8	Examples of diet tags	14
9	Allergen predicates	14
10	Examples of allergens	14
11	Diet compatibility structure	15
12	Diet compatibility rules	15
13	Allergen check structure	15
14	Rules for allergen detection	15
15	Calorie-conscious filter rule	16
16	Allergen safety check	16
17	Full compatibility rule	16
18	List of compatible meals	16
19	Meal + type compatibility	17
20	List of meals with type info	17
21	Example usage	17
22	HermiT error with SWRL built-ins	23
23	Allergy-based exclusion rule	23
24	Calorie-conscious exclusion rule	24
25	Heavy meal tagging rule	24
26	Light meal tagging rule	25
27	Vegetarian diet exclusion rule	25
28	AnimalIngredientShape	25
29	MealIngredientShape	25
30	MealShape	26
31	GuestNameShape	26
32	GuestShape	26
33	pySHACL Validation Command	27
34	Validation Output	27
35	SPARQL Prefixes	27

List of Figures

1	Example of meals with attributes	7
2	Example of meals with attributes	7
3	Example of Input Data Type in the List of Meals Input	9
4	Overall structure of the Decision Model showing the interaction between filters and recommendation logic.	11
5	Core class hierarchy defined in Protégé.	18
6	Meal Class Overview, with Objects Properties, Data Properties and Individuals related	20
7	Overview of the individual Boscaiola , with annotated data and object properties.	20
8	SWRLRule Overview	24
9	BPMN model using AOAME for guest preference integration	31
10	BPMN model using Camunda	32
11	Jena Fuseki query	33
12	Semantic annotation of the task Create personalized Guest Menu using AOAME. Parameters like DietType , CaloriePreference , and Allergen are attached to enable semantic reasoning.	34
13	Jena Fuseki results	35

1 Introduction

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

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

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

2 Project Objective

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

To achieve this, the project will:

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

All related files, implementation details, and project documentation are publicly accessible on GitHub at the following link: https://github.com/HertzKaa/Kebe_exam_23-24.

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

3 Menu Structure and Testing Dataset

As part of the “Personalized Menu” project, a complete menu has been created, consisting of **76 items**, reflecting typical dishes of an Italian restaurant. Each menu item is described with multiple attributes including: **Name**, **Category**, **Calories**, and a series of boolean fields (1 or 0) that indicate the presence or absence of specific ingredients or allergens such as **Meat**, **Dairy**, **Eggs**, **Gelatin**, **Gluten**, **Nuts**, **Sulfites**, **Shellfish**, **Fish**, **Spicy**, and **Soy**. Additional attributes specify dietary suitability, such as whether an item is **Vegan**, **Vegetarian** or **Gluten-Free**.

The **Category** attribute plays an important role, as it groups similar items together, which reflects how a real menu would logically organize options for easier navigation, for example by grouping starters, pizzas, pasta dishes, and desserts. Furthermore, fields like **Ingredients**, **Tags**, and **Allergens** provide extra context for detailed filtering and personalized recommendations based on guest profiles.

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

Name	Category	Calories	Meat	Vegan	Vegetarian	Dairy	Eggs	Gelatin	Gluten	Nuts	Sulfites	Shellfish	Fish	Spicy	Soy	Ingredients	Tags	Allergens
Tiramisu	Dessert	450	0	0	1	1	1	0	1	0	0	0	0	0	0	0 Mascarpone, eggs, sugar, ladyfingers, espresso, cocoa powder	Vegetarian	Dairy, Eggs, Gluten
Cannoli Siciliani	Dessert	400	0	0	1	1	1	0	1	0	0	0	0	0	0	0 Ricotta, sugar, chocolate chips, candied fruit, fried pastry shell	Vegetarian	Dairy, Eggs, Gluten
Torta Caprese	Dessert	380	0	0	1	1	1	0	0	1	0	0	0	0	0	0 Dark chocolate, almonds, butter, sugar, eggs	Gluten-Free, Vegetarian	Nuts, Eggs, Dairy
Panna Cotta	Dessert	300	0	0	1	1	0	1	0	0	0	0	0	0	0	0 Cream, sugar, vanilla, gelatin, berry coulis	Gluten-Free, Vegetarian	Dairy, Gelatin

Figure 1: Example of meals with attributes

In addition to the main menu, a second dataset named **Menu_test** has been developed. This testing table includes all possible combinations of dietary preferences, allergens, and potential null values, specifically designed to rigorously test and debug the Decision Tables. The **Menu_test** list ensures that the recommendation logic works as expected across all edge cases, guaranteeing reliability and correctness of the filtering process implemented with Decision Tables and other knowledge-based solutions.

Name	Category	Calories	Carnivore	Vegan	Vegetarian	Dairy	Eggs	Gelatin	Gluten	Nuts	Sulfites	Shellfish	Fish	Soy
Carnivore - No Dairy - C.C.	Dessert	0	1	0	0	0	1	1	1	1	1	1	1	1
Carnivore - No Eggs - C.C.	Drinks	0	1	0	0	1	0	1	1	1	1	1	1	1
Carnivore - No Gelatine - C.C.	Pizza No Gluten	0	1	0	0	1	1	0	1	1	1	1	1	1
Carnivore - No Gluten - C.C.	Pizza	0	1	0	0	1	1	1	0	1	1	1	1	1
Carnivore - No Nuts - C.C.	Primi piatti	0	1	0	0	1	1	1	1	0	1	1	1	1

Figure 2: Example of meals with attributes

4 Decision Model: Trisotech

The **Trisotech Decision Modeler** is a cloud-based tool designed for modeling and implementing decision logic using the **Decision Model and Notation (DMN)** standard. DMN is an internationally recognized standard developed by the Object Management Group (OMG) for representing complex decision-making processes in a clear, structured, and technology-independent way.

With Trisotech Decision Modeler, users can create **Decision Requirement Diagrams (DRDs)** that visually map out the relationships between different decisions, input data, and business knowledge. The tool also allows for the definition of detailed **Decision Tables**, which are ideal for structuring conditional logic in an easy-to-read, tabular format. This makes it particularly suitable for representing rules related to personalized meal recommendations, where multiple variables such as allergens, dietary preferences, and calorie constraints must be considered simultaneously.

One of the key advantages of using Trisotech is its accessibility for both technical and non-technical stakeholders. Its intuitive graphical interface allows restaurant managers or other domain experts to understand and, if necessary, modify the decision logic without requiring deep technical expertise. This aligns perfectly with the project’s goal of creating a system that is not only technically robust but also easy to apply in real-world restaurant environments.

4.1 Decision Modeler

Our implementation within the **Trisotech Decision Modeler** is entirely based on the use of an external **XML file** containing all the dishes and their related attributes. This design choice allows the system to remain **highly modular, scalable, and easily maintainable**, as the menu data can be modified or expanded without the need to alter the core decision logic.

While Trisotech traditionally focuses on the use of standard **Decision Tables**, we opted for a more flexible and advanced approach by integrating scripts within the decision model. This scripting capability enabled us to create **decision models that are much closer to real-world scenarios**, capable of processing large datasets like our comprehensive 76-item menu, including complex attributes such as allergens, dietary preferences, and ingredient details.

This approach provides several key advantages:

- **Scalability:** New dishes or attributes can be added to the XML file without requiring structural changes to the decision logic.
- **Resilience to Errors:** By externalizing the data, we minimize the risk of inconsistencies or manual errors in the decision logic itself.
- **Enhanced Realism:** The decision models handle real-world complexities such as combined dietary restrictions, overlapping allergens, and personalized recommendations, beyond what traditional static decision tables typically support.
- **Additional Features:** Integration of scripts allowed us to implement user-friendly features such as dynamically displaying the complete list of available dishes directly within the decision environment, improving transparency for stakeholders.

Although it would have been possible to rely solely on traditional decision tables, this would have significantly limited the project in terms of flexibility and realism. Static tables are effective for simple, predefined logic, but they struggle with dynamic, data-driven scenarios like personalized menu filtering, where the dataset may frequently change or expand.

By incorporating external XML files and leveraging the advanced capabilities of Trisotech’s scripting environment, we developed a solution that is both **technically robust and operationally aligned** with the needs of a real restaurant environment, ensuring adaptability and long-term maintainability.

4.2 Elements of the Decision Diagram

The **Decision Requirement Diagram (DRD)** developed within Trisotech Decision Modeler consists of a set of key inputs and decisions structured entirely around **custom data types**, designed to ensure consistency, scalability, and an intuitive user experience. The use of custom types allows for functionalities such as reading structured table data, offering drop-down menus, and consistently processing the same type of input across the system.

4.2.1 Input Data

The primary inputs for the decision model are:

- **List of Meals:** Represents the complete dataset of dishes. Defined with the custom type `ListOfMeals`, reflecting the XML structure with fields like meal name, category, calories, and dietary/allergen indicators. Modeled as a *collection*.
- **Guest Allergies:** Captures guest-specified allergies. Also a *collection*, enabling multiple allergy inputs (e.g., Gluten, Dairy).
- **Guest Diet:** A *single-value input* representing dietary preference (e.g., Carnivore, Vegetarian, Vegan).
- **Guest Calories:** A boolean *single-value input* indicating whether the guest is calorie-conscious.
- **Show Full Menu?:** A boolean input to optionally bypass filtering and display the full menu.

The screenshot shows a 'Data Type' configuration window for a custom type named 'ListOfMeals'. The window has a title bar with 'Data Type' and standard window controls. On the left, there is a sidebar with the name 'ListOfMeals' and a tree icon. The main area displays a table with 15 rows, each representing a field in the data type. Each row has a number, a field name, a dropdown menu for the data type, and a link icon. The fields and their configured data types are: 1. Name (Text), 2. Vegan (Number), 3. Vegetarian (Number), 4. Carnivore (Number), 5. Dairy (Number), 6. Eggs (Number), 7. Gelatin (Number), 8. Gluten (Number), 9. Nuts (Number), 10. Sulfites (Number), 11. Shellfish (Number), 12. Fish (Number), 13. Soy (Number), 14. Calories (Number), and 15. Category (Text). At the bottom of the table is a plus sign icon. Below the table are 'Save' and 'Cancel' buttons. At the very bottom of the window is a 'Close' button.

Index	Field Name	Data Type
1	Name	Text
2	Vegan	Number
3	Vegetarian	Number
4	Carnivore	Number
5	Dairy	Number
6	Eggs	Number
7	Gelatin	Number
8	Gluten	Number
9	Nuts	Number
10	Sulfites	Number
11	Shellfish	Number
12	Fish	Number
13	Soy	Number
14	Calories	Number
15	Category	Text

Figure 3: Example of Input Data Type in the List of Meals Input

4.2.2 Error Handling and Robustness

The system has been designed with careful attention to error handling and input validation. If the user provides incomplete, incorrect, or no input, the decision model defaults to showing the full menu or applies only the available filters. This prevents failures or inconsistent behavior while ensuring a user-friendly experience, even in cases where the input is partial or omitted altogether.

By leveraging custom data types, collections for multi-valued inputs, and robust error handling, the decision model achieves high levels of **realism, flexibility, and resilience**, making it well-suited for real-world restaurant operations where menu data and customer preferences vary frequently.

4.3 Decision Blocks in the DMN Model

In addition to the input data, the core structure of the DMN model is built around **Decision Blocks**, which forms the logical backbone of the meal recommendation process. The model includes **three dedicated decision blocks**, each designed to handle a specific filtering criterion:

- **Filter by Allergens**
- **Filter by Diet Type**
- **Filter by Calorie Preference**

4.3.1 Functionality of the Decision Blocks

Each block operates independently, applying a specific filter to the complete **List of Meals**, based on the corresponding user inputs:

- **Filter by Allergens:** excludes meals with ingredients matching guest allergies.
- **Filter by Diet Type:** removes meals not aligned with the declared dietary preference.
- **Filter by Calorie Preference:** removes meals exceeding a defined calorie threshold when the guest is calorie-conscious.

A key aspect of these decision blocks is the use of **FEEL expressions** (Friendly Enough Expression Language), the standard expression language defined by DMN specifications. FEEL expressions allow for precise, readable, and powerful logic within the decision tables and scripts, enabling operations such as:

- Evaluating conditions on collections of data.
- Filtering lists based on multiple attributes (e.g., allergens, diet compatibility).
- Performing intersections of lists across different decision blocks.
- Handling null or incomplete input gracefully through conditional logic.

By leveraging FEEL expressions, the model achieves a high degree of flexibility and expressiveness, essential for implementing realistic, data-driven decision logic that matches the complexity of real-world restaurant scenarios.

4.3.2 Main Block: Recommend Meals

All three filtering processes converge into the final decision block: **Recommend Meals**. This block aggregates the outputs of the previous filters by performing a logical **intersection** of the filtered lists, using FEEL expressions to ensure consistent handling of collections and conditions. The result is a refined, personalized menu containing only the dishes that meet **all specified preferences and restrictions**.

If any of the preceding decision blocks are bypassed due to missing or partial input, the intersection logic dynamically adapts, considering only the available filtering results. This ensures that the recommendation process remains functional under all input conditions.

Additionally, all blocks, including **Recommend Meals**, are designed to work with **collections**, enabling the model to handle multiple outputs at each stage. This structure mirrors real-world scenarios where multiple meals can pass or fail various filtering criteria simultaneously.

This design results in a decision model that is both compliant with project requirements and capable of managing complex, realistic filtering scenarios with a high degree of precision and adaptability.

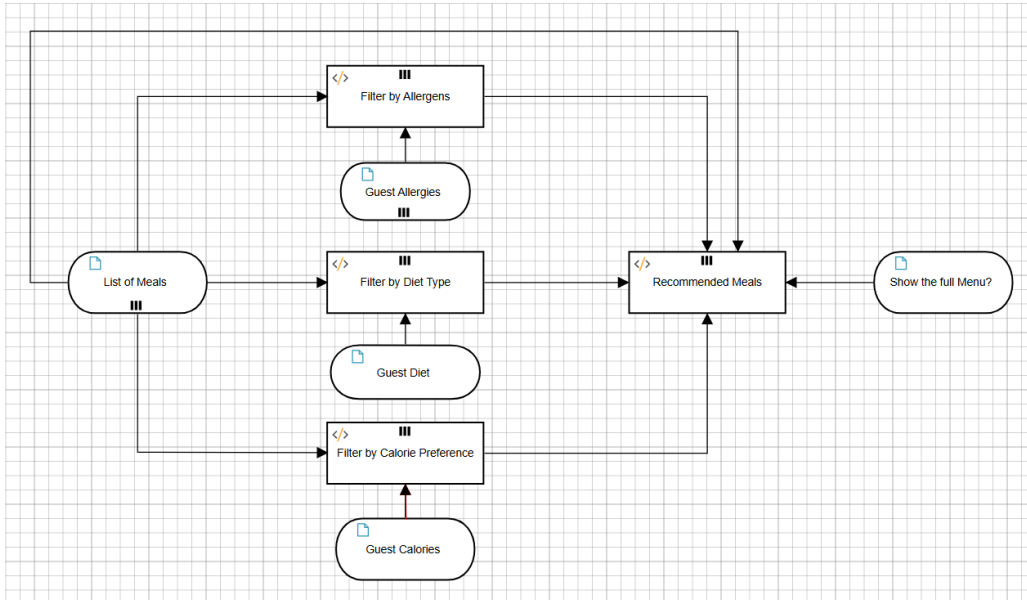


Figure 4: Overall structure of the Decision Model showing the interaction between filters and recommendation logic.

4.3.3 Testing and Validation

After implementing the DMN diagrams and decision logic, an extensive testing phase was conducted to ensure the system's correctness and robustness. The tests included both manual input scenarios and automated validation using the **Menu_test** dataset, which contains all possible combinations of allergens, dietary preferences, calorie-conscious profiles, and null values.

The results of the testing phase were fully successful. The decision blocks, powered by **FEEL expressions**, consistently generated accurate and reliable outputs. The recommended meal lists produced by the system were correctly filtered based on the provided inputs, with no errors or inconsistencies observed during the testing process.

All **decision tables** and associated logic behaved as expected, effectively handling both valid and edge-case inputs. The system demonstrated its ability to:

- Correctly exclude meals containing allergens specified by the guest.
- Respect dietary restrictions such as vegetarian, vegan, or carnivorous preferences.

- Filter meals based on calorie-conscious criteria when required.
- Handle incomplete or missing inputs gracefully without causing system failures.

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

5 SWISH Prolog

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

At its core, Prolog operates on the principles of:

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

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

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

5.1 Prolog in the Personalized Menu Project

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

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

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

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

5.2 Facts Definition in Prolog

The foundation of the knowledge base for the “Personalized Menu” project in **Prolog** is built upon a structured set of **facts**, which explicitly define the available meals, their categories, calorie content, dietary profiles, and allergen information. These facts allow the system to reason about meal compatibility, filtering options, and personalized recommendations for different guest profiles.

5.2.1 Meal Declaration

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

```
1 meal(MealName).
```

Listing 1: Meal declaration

For example:

```
1 meal(pizza_margherita).
2 meal(carbonara).
3 meal(vegan_salad).
```

Listing 2: Example of meal declaration

These facts simply register all the available dishes, forming the basis for further classification and reasoning.

5.2.2 Meal Categories

Each meal is assigned a **category**, which is useful for organizing and filtering dishes by type (e.g., pasta, pizza, dessert, main course). The format is:

```
1 meal_type(MealName, Category).
```

Listing 3: Meal categories

Example:

```
1 meal_type(pizza_margherita, pizza).
2 meal_type(carbonara, pasta).
3 meal_type(vegan_salad, salad).
```

Listing 4: Example of categorized meals

This categorization reflects how real menus are structured, making the system more realistic and practical.

5.2.3 Calorie Information

The energy content of each meal, measured in kilocalories (kcal), is defined using:

```
1 calories(MealName, CalorieValue).
```

Listing 5: Calorie structure

Example:

```
1 calories(pizza_margherita, 800).
2 calories(vegan_salad, 350).
```

Listing 6: Example of calories per meal

This information is critical for filtering meals based on calorie-conscious preferences.

5.2.4 Dietary Tags and Allergen Indicators

The system defines both dietary profiles and allergen content using dedicated facts. These indicators allow the system to filter meals in accordance with guest dietary needs and allergy constraints.

Dietary Tags

```
1 carnivore(MealName).      % Indicates meal contains meat or is suitable
   for carnivores
2 vegetarian(MealName).    % Indicates meal is vegetarian-friendly
3 vegan(MealName).        % Indicates meal is entirely plant-based
```

Listing 7: Dietary tags

Example:

```
1 vegetarian(pizza_margherita).
2 carnivore(carbonara).
3 vegan(vegan_salad).
```

Listing 8: Examples of diet tags

These tags correspond to boolean values derived from the menu dataset, where a "1" indicates that the tag applies to the meal.

Allergen Tags

Allergen presence is defined with specific facts, ensuring that the system can exclude meals containing ingredients that may trigger guest allergies. The format is:

```
1 contains_dairy(MealName).
2 contains_eggs(MealName).
3 contains_gelatin(MealName).
4 contains_gluten(MealName).
5 contains_nuts(MealName).
6 contains_sulfites(MealName).
7 contains_shellfish(MealName).
8 contains_fish(MealName).
9 contains_soy(MealName).
```

Listing 9: Allergen predicates

Example:

```
1 contains_dairy(pizza_margherita).
2 contains_gluten(carbonara).
3 contains_nuts(vegan_salad).
```

Listing 10: Examples of allergens

The presence of each allergen is again based on boolean values in the dataset, where "1" confirms the allergen's presence in the meal.

6 Prolog Rules: Meal Compatibility and Recommendation Logic

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

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

6.1 Diet Compatibility Rules

The system defines whether a meal complies with the guest's dietary preference through the predicate:

```
1 diet_compatible(Meal, DietType).
```

Listing 11: Diet compatibility structure

Supported diet types:

- vegan
- vegetarian
- carnivore

Rule logic:

```
1 diet_compatible(Meal, vegan) :-  
2     vegan(Meal).  
3  
4 diet_compatible(Meal, vegetarian) :-  
5     vegetarian(Meal);  
6     vegan(Meal). % Vegans also meet vegetarian requirements  
7  
8 diet_compatible(Meal, carnivore) :-  
9     carnivore(Meal).
```

Listing 12: Diet compatibility rules

This ensures:

- Vegan meals satisfy only vegan guests.
- Vegetarian guests accept both vegetarian and vegan meals.
- Carnivore meals are suitable for guests with no dietary restrictions.

6.2 Allergen Checks

To manage allergen-related filtering, the system uses the `meal_contains_allergen/2` predicate:

```
1 meal_contains_allergen(Meal, Allergen).
```

Listing 13: Allergen check structure

Example mapping:

```
1 meal_contains_allergen(Meal, dairy)      :- contains_dairy(Meal).  
2 meal_contains_allergen(Meal, eggs)       :- contains_eggs(Meal).  
3 meal_contains_allergen(Meal, gelatin)    :- contains_gelatin(Meal).  
4 meal_contains_allergen(Meal, gluten)     :- contains_gluten(Meal).  
5 meal_contains_allergen(Meal, nuts)       :- contains_nuts(Meal).  
6 meal_contains_allergen(Meal, sulfites)   :- contains_sulfites(Meal).  
7 meal_contains_allergen(Meal, shellfish)  :- contains_shellfish(Meal).  
8 meal_contains_allergen(Meal, fish)       :- contains_fish(Meal).  
9 meal_contains_allergen(Meal, soy)        :- contains_soy(Meal).
```

Listing 14: Rules for allergen detection

This approach abstracts allergen detection, allowing for easy extension and uniform filtering logic regardless of the allergen type.

6.3 Calorie-Conscious Meal Filter

Guests who are calorie-conscious can request meals under a specific calorie threshold, defined as:

```
1 calorie_conscious_meal(Meal) :-  
2     calories(Meal, Calories),  
3     Calories =< 600.
```

Listing 15: Calorie-conscious filter rule

This rule filters meals based on a 600 kcal limit, aligning with typical calorie-conscious guidelines.

6.4 Overall Meal Compatibility Checks

The system integrates dietary, allergen, and calorie preferences using the following rules:

6.4.1 meal_safe_for_allergies/2

Determines if a meal is safe given the list of guest allergies:

```
1 meal_safe_for_allergies(_, []).  
2 meal_safe_for_allergies(Meal, [Allergen | Rest]) :-  
3     \+ meal_contains_allergen(Meal, Allergen),  
4     meal_safe_for_allergies(Meal, Rest).
```

Listing 16: Allergen safety check

This recursive rule ensures that none of the guest's allergies are present in the meal.

6.4.2 compatible_meal/4

Checks full compatibility for a meal based on diet, allergies, and calorie-conscious preference:

```
1 compatible_meal(Meal, Diet, Allergies, CalorieConscious) :-  
2     diet_compatible(Meal, Diet),  
3     meal_safe_for_allergies(Meal, Allergies),  
4     (    CalorieConscious = true -> calorie_conscious_meal(Meal)  
5     ;    true % If calorie-conscious filter is not active, include all  
6           calories  
       ).
```

Listing 17: Full compatibility rule

6.4.3 compatible_meals/4

Finds all meals that satisfy the specified constraints:

```
1 compatible_meals(Diet, Allergies, CalorieConscious, Meals) :-  
2     findall(Meal, compatible_meal(Meal, Diet, Allergies,  
3     CalorieConscious), Meals).
```

Listing 18: List of compatible meals

6.5 Meal Compatibility with Meal Type

To enhance filtering with additional meal categorization (e.g., pizza, pasta, dessert), the following predicates extend compatibility logic:

6.5.1 compatible_meal_with_type/5

```
1 compatible_meal_with_type(Meal, Diet, Allergies, CalorieConscious, Type)
   :-
2   compatible_meal(Meal, Diet, Allergies, CalorieConscious),
3   meal_type(Meal, Type).
```

Listing 19: Meal + type compatibility

6.5.2 compatible_meals_with_type/4

```
1 compatible_meals_with_type(Diet, Allergies, CalorieConscious,
   MealsWithTypes) :-
2   findall((Meal, Type), compatible_meal_with_type(Meal, Diet,
   Allergies, CalorieConscious, Type), MealsWithTypes).
```

Listing 20: List of meals with type info

Example Query:

```
1 compatible_meals_with_type(vegan, [gluten], true, Meals).
```

Listing 21: Example usage

This returns a list of tuples (**Meal**, **Type**) representing all vegan, gluten-free, calorie-conscious meals along with their category, making the output more informative for practical use.

6.6 Ontology Engineering Role

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

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

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

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

7 Knowledge Representation Technologies

7.1 Turtle Syntax

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

What makes Turtle particularly suitable for ontology engineering is its ability to define **prefixes** that abbreviate long **URIs (Uniform Resource Identifiers)**. This feature significantly improves both the readability and maintainability of the ontology, especially when compared to more verbose alternatives like RDF/XML. By saving our ontology in a `.ttl` file, we ensured that the structure of the knowledge base remained both **clear and accessible**, facilitating the modeling of classes, properties, and individuals aligned with the personalized menu domain.

7.2 Protégé

The ontology was developed using **Protégé**, a widely used open-source tool for ontology modeling developed by **Stanford University**. Protégé provides a robust and user-friendly environment for creating, editing, and managing ontologies. It supports a range of ontology languages, including **OWL (Web Ontology Language)** and **RDF (Resource Description Framework)**. Using Protégé, we were able to formally define domain knowledge through the creation of **classes**, **object properties**, **data properties**, and **individuals**.

7.3 Our Ontology

Our ontology is structured in a clear and well-organized hierarchy, which serves as the foundation for a consistent and scalable development of the project. By logically categorizing key concepts—such as ingredients, allergens, dietary types, and guest preferences—it ensures that the domain knowledge is represented in a coherent and semantically meaningful way.

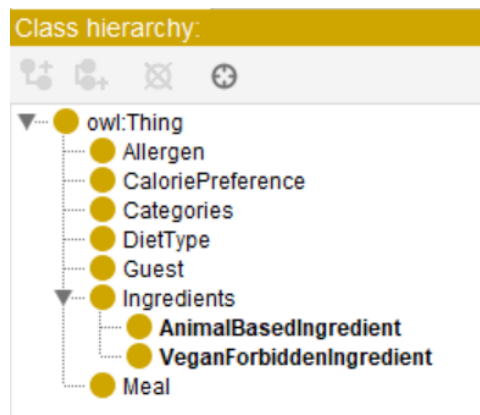


Figure 5: Core class hierarchy defined in Protégé.

This structured approach is essential not only for maintaining clarity during the modeling phase, but also for enabling effective reasoning, inference, and personalized decision-making throughout the system.

7.4 Classes

As previously introduced, the figure highlights the structure of the ontology’s class hierarchy. Two core classes can be identified: **Guest** and **Meal**, representing, respectively, the user who

interacts with the system and the dish offered by the menu. However, in order to determine which meals are suitable for a specific guest, it is essential to introduce additional parameters that enable user profiling and the identification of dietary preferences or restrictions.

To address this, the ontology includes several supporting classes such as **Allergen**, **CaloriePreference**, **Categories**, **DietType**, and **Ingredients**. The **Ingredients** class, in particular, is further refined through two subclasses: **AnimalBasedIngredient** and **VeganForbiddenIngredient**, allowing the system to capture more nuanced dietary constraints.

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

7.4.1 Meal Class

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

Object properties:

- **ContainsAllergen**: links the meal to allergens it contains.
- **HasIngredient**: connects a meal to the ingredients it is composed of.
- **hasCategory**: assigns the meal to a category such as "First Serve" or "Drink".
- **notRecommendedFor**: indicates which guests should avoid a particular meal based on their profile.

Data properties:

- **Calories**: associates each instance of the **Meal** class with a numerical value (typically an integer or float) representing its caloric content.
- **MealName**: links a meal to a string literal representing its human-readable name.
- **isHeavyMeal**: a boolean data property used to indicate whether a given meal exceeds a predefined caloric threshold (e.g., 800 kcal).
- **isLightMeal**: a boolean data property indicating whether a meal falls below a certain caloric value (e.g., 400 kcal).

Examples of individuals:

- *Acqua_Naturale_Frizzante*, *Aperol_Spritz*, *Boscaiola*, *Buffalina*, *Cappuccino*, *Caffè_Espresso*

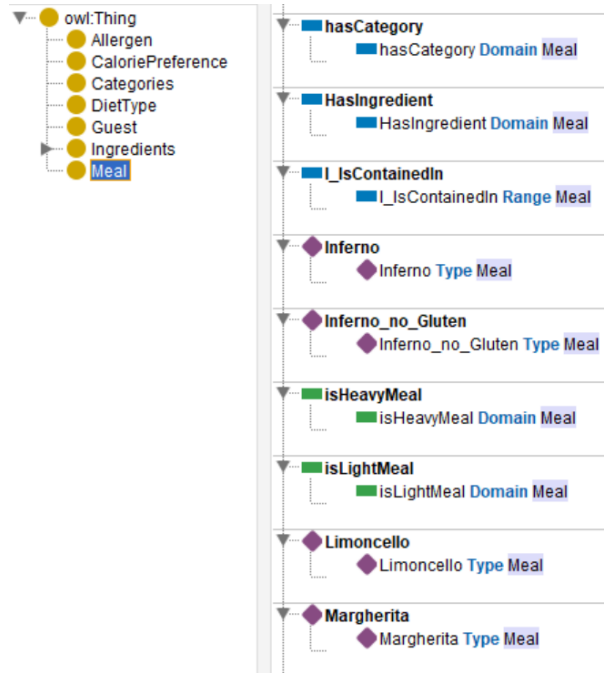


Figure 6: Meal Class Overview, with Objects Properties, Data Properties and Individuals related

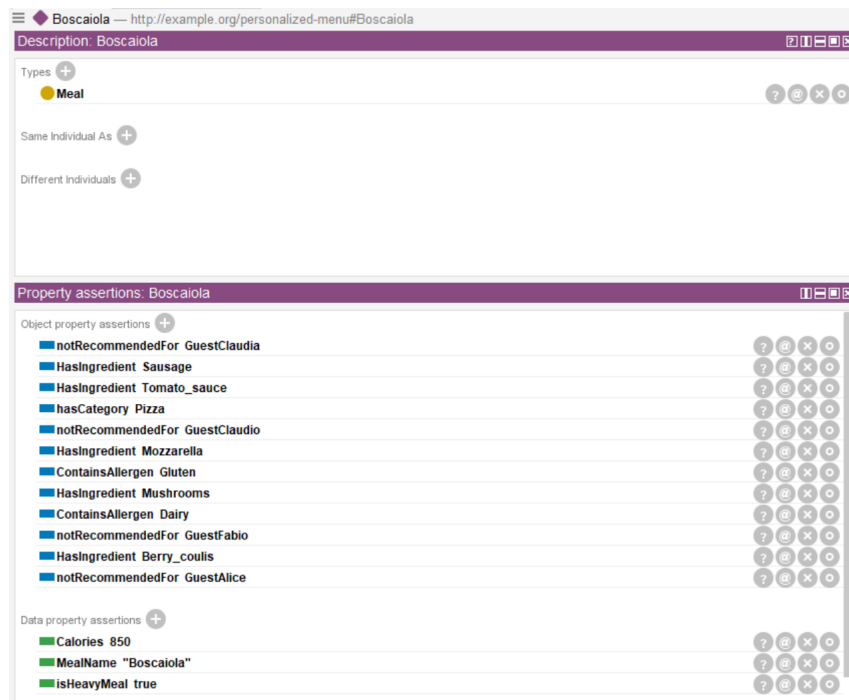


Figure 7: Overview of the individual Boscaiola, with annotated data and object properties.

7.4.2 Guest Class

The **Guest** class represents the users of the system who receive personalized meal recommendations. Each individual belonging to this class is described by a set of properties that capture their dietary preferences, caloric awareness, and possible allergies. These characteristics serve as the basis for filtering and determining suitable meals.

Object properties:

- **hasAllergy**: links a guest to one or more instances of the **Allergen** class.
- **hasCaloriePreference**: connects the guest to their caloric sensitivity profile (**CalorieConscious** or **NotCalorieConscious**).
- **hasDietType**: specifies the dietary regime of the guest, such as **vegan** or **carnivore**.

Data properties:

- **GuestName**: stores the name or label of the guest individual.

Examples of individuals:

- **GuestAlice**, **GuestClaudia**, **GuestClaudio**, **GuestFabio**

7.4.3 Ingredients Class

The **Ingredients** class models the components used in the preparation of meals. It plays a key role in reasoning over dietary restrictions and is further refined through specialized subclasses.

Subclasses:

- **AnimalBasedIngredient**: identifies ingredients derived from animals.
- **VeganForbiddenIngredient**: captures ingredients that violate vegan dietary rules.

Object properties:

- **HasIngredient**: connects meals to ingredients.

Data properties:

- **IngredientName**: denotes the name of the ingredient.

Examples of individuals:

- **Basil**, **Mozzarella**, **Anchovies**, **Tuna**, **Tomato_rice**

7.4.4 DietType Class

The **DietType** class defines the dietary regimes that guests may follow. It enables the ontology to support reasoning about whether a specific meal aligns with the guest's dietary needs.

Object properties:

- **hasDietType**: links a guest to their declared dietary type.

Examples of individuals:

- **Carnivore**, **Vegan**, **Vegetarian**

7.4.5 Categories Class

The **Categories** class organizes meals into high-level semantic groups. This classification supports filtering and structuring of meals in the recommendation system.

Object properties:

- **hasCategory**: connects meals to a specific category.

Examples of individuals:

- **Dessert**, **Drink**, **Pizza**, **First_Serve**, **Second_Serve**, **Pizza_No_Gluten**

7.4.6 CaloriePreference Class

The **CaloriePreference** class models a guest’s sensitivity to caloric intake. It is used to infer whether certain meals should be avoided based on their energy content.

Object properties:

- **hasCaloriePreference**: connects a guest to either a **CalorieConscious** or **NotCalorieConscious** profile.

Examples of individuals:

- **CalorieConscious**, **NotCalorieConscious**

7.4.7 Allergen Class

The **Allergen** class represents known food allergens that may be present in meals or ingredients. This class is central to managing food safety and ensuring the exclusion of incompatible meals for allergic guests.

Object properties:

- **hasAllergy**: connects a guest to allergens they must avoid.
- **ContainsAllergen**: links meals to the allergens they contain.

Examples of individuals:

- **Dairy**, **Gluten**, **Eggs**, **Nuts**, **Shellfish**, **Soy**, **Sulfites**, **Gelatin**, **Fish**

In addition to the direct object properties used to associate meals with their ingredients and allergens, the ontology also defines **inverse object properties** to enable bidirectional reasoning. Specifically, the properties **HasIngredient** and **ContainsAllergen**, which link a **Meal** to its constituent **Ingredients** and **Allergens**, have corresponding inverses named **I_IsContainedIn** and **A_IsContainedIn**, respectively.

7.5 SWRL Rules

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

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

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

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

Initially, we used the **HermiT 1.4.3.456** reasoner; however, due to compatibility issues with SWRL built-ins such as **swrlb:greaterThan**, we migrated to the **Pellet** reasoner, which supports numerical comparisons and datatype reasoning more robustly. The effects of SWRL reasoning are visible only when the ontology is actively classified using Pellet.

7.5.1 Built-in Atom Limitation in HermiT

Initially, we attempted to use the **HermiT 1.4.3.456** reasoner for reasoning tasks. However, HermiT currently **does not support SWRL built-in atoms**, such as `swrlb:greaterThan` or `swrlb:lessThanOrEqual`. This limitation resulted in a runtime failure during classification, with the following representative error:

```
1 An error occurred during reasoning: A SWRL rule uses a built-in atom,  
  but built-in atoms are not supported yet.  
2 java.lang.IllegalArgumentException: A SWRL rule uses a built-in atom,  
  but built-in atoms are not supported yet.
```

Listing 22: HermiT error with SWRL built-ins

Due to this limitation, we switched to the **Pellet** reasoner, which offers full support for SWRL built-in atoms and datatype comparisons. With Pellet, it was possible to evaluate rules involving numerical thresholds (e.g., calories), boolean flags, and conditional exclusion logic. It is important to note that the results of SWRL reasoning are **not persisted** in the ontology by default, but are **visible only during the execution of the reasoner**.

Below is a detailed explanation of each SWRL rule implemented in our ontology.

7.5.2 1. Allergy Check

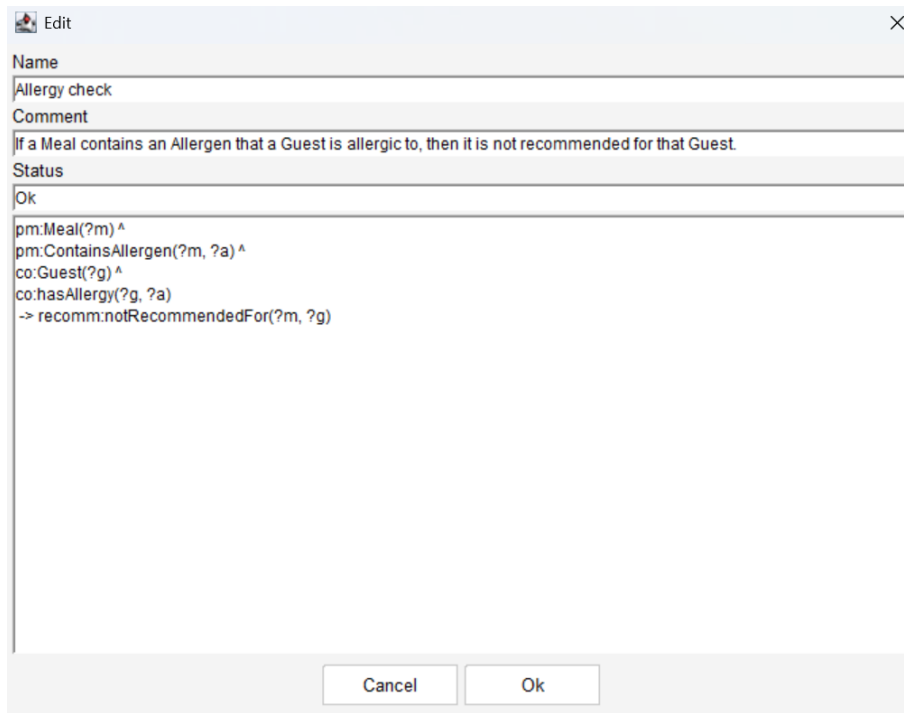
Rule:

```
1 pm:Meal(?m) ^  
2 pm:ContainsAllergen(?m, ?a) ^  
3 co:Guest(?g) ^  
4 co:hasAllergy(?g, ?a)  
5 -> recomm:notRecommendedFor(?m, ?g)
```

Listing 23: Allergy-based exclusion rule

Description: If a **Meal** contains an **Allergen** to which a **Guest** is allergic, then that meal is inferred to be `notRecommendedFor` for that guest.

Use case: If **GuestClaudia** is allergic to **Dairy**, and **Cannoli_Siciliani** contains **Dairy**, then this meal will be excluded from her options.



The dialog box, titled 'Edit', contains the following fields:

- Name:** Allergy check
- Comment:** If a Meal contains an Allergen that a Guest is allergic to, then it is not recommended for that Guest.
- Status:** Ok

The main text area contains the following SWRL rule:

```
pm:Meal(?m) ^
pm:ContainsAllergen(?m, ?a) ^
co:Guest(?g) ^
co:hasAllergy(?g, ?a)
-> recomm.notRecommendedFor(?m, ?g)
```

At the bottom are 'Cancel' and 'Ok' buttons.

Figure 8: SWRLRule Overview

7.5.3 2. Calorie-Based Recommendation

```
1 pm:Calories(?m, ?c) ^
2 swrlb:greaterThan(?c, 600) ^
3 pm:Meal(?m) ^
4 co:Guest(?g) ^
5 co:hasCaloriePreference(?g, co:CalorieConscious)
6 -> recomm.notRecommendedFor(?m, ?g)
```

Listing 24: Calorie-conscious exclusion rule

Description: If a **Meal** contains more than 600 kcal and the **Guest** is marked as **CalorieConscious**, then that meal is flagged as not recommended.

Use case: A calorie-conscious guest will avoid high-calorie dishes like **Bistecca_alla_Fiorentina**.

7.5.4 3. Heavy Meal Definition

```
1 pm:Meal(?m) ^
2 pm:Calories(?m, ?c) ^
3 swrlb:greaterThan(?c, 800)
4 -> pm:isHeavyMeal(?m, true)
```

Listing 25: Heavy meal tagging rule

Description: Assigns a boolean classification to meals exceeding a caloric threshold of 800 kcal.

Use case: Meals like **Buffalina** will be tagged as **isHeavyMeal = true**.

7.5.5 4. Light Meal Definition


```

1 pm:Meal(?m) ^
2 pm:Calories(?m, ?c) ^
3 swrlb:lessThanOrEqual(?c, 400)
4 -> pm:isLightMeal(?m, true)

```

Listing 26: Light meal tagging rule

Description: Marks a meal as light if its caloric value is 400 kcal or less.

Use case: If `Zuppa_di_Farro_e_Legumi` has 350 kcal, it will be inferred as a light meal.

7.5.6 5. Vegetarian Diet Recommendation

```

1 co:hasDietType(?g, co:Vegetarian) ^
2 pm:Meal(?m) ^
3 pm:AnimalBasedIngredient(?i) ^
4 co:Guest(?g) ^
5 pm:HasIngredient(?m, ?i)
6 -> recomm:notRecommendedFor(?m, ?g)

```

Listing 27: Vegetarian diet exclusion rule

Description: Enforces vegetarian constraints: meals with `AnimalBasedIngredient` are excluded for vegetarian guests.

Use case: A vegetarian guest will not be recommended dishes containing meat, fish, or animal-derived ingredients.

7.6 SHACL Shapes

SHACL (Shapes Constraint Language) is a W3C standard used to validate RDF data by defining structural and logical constraints, known as *shapes*. These shapes are themselves RDF graphs that describe expectations over the data model. They are essential in ensuring data consistency, enforcing modeling guidelines, and preventing semantic errors during ontology usage.

In our ontology, SHACL shapes were employed to validate both the structure and semantics of classes such as `Meal`, `Ingredient`, and `Menu`, as well as to verify classification-related conditions (e.g., vegetarian, ingredient or caloric content). This validation step ensures that the data instances conform to the logical rules defined in the ontology and are suitable for reasoning and recommendation.

7.6.1 AnimalIngredientShape

```

1 pm:AnimalIngredientShape a sh:NodeShape ;
2   sh:class pm:Ingredients ;
3   sh:targetClass pm:AnimalBasedIngredient .

```

Listing 28: AnimalIngredientShape

Description: Ensures that every `AnimalBasedIngredient` is a valid instance of the broader `Ingredients` class.

7.6.2 MealIngredientShape

```

1 pm:MealIngredientShape a sh:NodeShape ;
2   sh:property [
3     sh:minCount 1 ;
4     sh:path pm:HasIngredient

```

```

5     ] ;
6     sh:targetClass pm:Meal .

```

Listing 29: MealIngredientShape

Description: Verifies that each `Meal` individual has at least one associated ingredient.

7.6.3 MealShape

```

1 pm:MealShape a sh:NodeShape ;
2   sh:property [
3     sh:datatype xsd:integer ;
4     sh:maxCount 1 ;
5     sh:minCount 1 ;
6     sh:minInclusive 0 ;
7     sh:path pm:Calories
8   ] ;
9   sh:targetClass pm:Meal .

```

Listing 30: MealShape

Description: Ensures that calorie values for meals are non-negative integers, and that each meal has exactly one calorie declaration.

7.6.4 GuestNameShape

```

1 co:GuestNameShape a sh:NodeShape ;
2   sh:property [
3     sh:datatype xsd:string ;
4     sh:minCount 1 ;
5     sh:path co:GuestName
6   ] ;
7   sh:targetClass co:Guest .

```

Listing 31: GuestNameShape

Description: Validates that each guest has a name specified as a string.

7.6.5 GuestShape

```

1 co:GuestShape a sh:NodeShape ;
2   sh:or_ (
3     [ sh:property [ sh:minCount 1 ; sh:path co:hasDietType ] ]
4     [ sh:property [ sh:minCount 1 ; sh:path co:hasAllergy ] ]
5   ) ;
6   sh:targetClass co:Guest .

```

Listing 32: GuestShape

Description: Requires each guest to have at least one descriptor: either a diet type or an allergy.

7.6.6 SHACL Validation Process

Once all SHACL shapes were defined, we validated our RDF dataset to ensure that all individuals and their associated properties conformed to the constraints.

We used the `pySHACL` tool, which allows command-line validation with support for multiple entailment regimes.

Command used:

```

1 pyshacl -d ontology_PersonalizedMenu.ttl -s shapes_PersonalizedMenu.ttl
  -m -i owlrl -df turtle

```

Listing 33: pySHACL Validation Command

Where:

- `-d` specifies the data graph
- `-s` specifies the SHACL shapes graph
- `-m` enables SHACL advanced features
- `-i owlrl` enables OWL 2 RL inferencing
- `-df turtle` sets RDF format to Turtle

Validation result:

```

1 Validation Report
2 Conforms: True

```

Listing 34: Validation Output

This confirms that our ontology conforms to all defined SHACL constraints, validating its correctness and reliability for reasoning tasks.

7.7 SPARQL Queries

SPARQL (SPARQL Protocol and RDF Query Language) is a W3C-standardized language used to query and manipulate data in RDF format. Within our project, SPARQL was essential for retrieving information, validating reasoning, and inspecting class assignments and relationships.

Queries were executed through **Apache Jena Fuseki**, a SPARQL endpoint server. The following prefixes were used to simplify and shorten query syntax:

```

1 PREFIX co: <http://www.co-ode.org/ontologies/ont.owl#>
2 PREFIX pm: <http://example.org/personalized-menu#>
3 PREFIX recomm: <http://www.semanticweb.org/fabio/ontologies/2025/5/
  PersonalizedMenu#>

```

Listing 35: SPARQL Prefixes

7.7.1 1. Allergy-based Meal Filtering

```

1 SELECT ?meal ?guest
2 WHERE {
3   ?meal recomm:notRecommendedFor ?guest .
4   ?guest co:hasAllergy ?allergen .
5   ?meal pm:ContainsAllergen ?allergen .
6 }

```

Returns meals excluded due to allergen conflicts.

7.7.2 2. Calorie-Conscious Guests – High-Calorie Meal Filtering

```
1 SELECT ?meal ?guest (STR(?calories) AS ?calorieValue)
2 WHERE {
3     ?meal recomm:notRecommendedFor ?guest .
4     ?guest co:hasCaloriePreference co:CalorieConscious .
5     ?meal pm:Calories ?calories .
6     FILTER(?calories > 600)
7 }
```

Filters meals over 600 kcal for guests with calorie sensitivity.

7.7.3 3. Vegetarian Guests – Animal-Based Ingredient Exclusion

```
1 SELECT ?meal ?guest ?ingredient
2 WHERE {
3     ?meal recomm:notRecommendedFor ?guest .
4     ?guest co:hasDietType co:Vegetarian .
5     ?meal pm:HasIngredient ?ingredient .
6     ?ingredient a pm:AnimalBasedIngredient .
7 }
```

Detects incompatibility with vegetarian diet.

7.7.4 4. Recommended Meals for a Specific Guest

```
1 SELECT ?meal
2 WHERE {
3     ?meal a pm:Meal .
4     FILTER NOT EXISTS {
5         ?meal recomm:notRecommendedFor co:GuestClaudio .
6     }
7 }
```

Finds meals that are suitable for GuestClaudio.

7.7.5 5. Explain Why a Meal is Not Recommended

```
1 SELECT ?meal ?guest ?reason
2 WHERE {
3     ?meal recomm:notRecommendedFor ?guest .
4
5     OPTIONAL {
6         ?guest co:hasAllergy ?reason .
7         ?meal pm:ContainsAllergen ?reason .
8     }
9
10    OPTIONAL {
11        ?guest co:hasCaloriePreference co:CalorieConscious .
12        ?meal pm:Calories ?cal .
13        FILTER(?cal > 600)
14        BIND("Too many calories" AS ?reason)
15    }
16
17    OPTIONAL {
18        ?guest co:hasDietType co:Vegetarian .
19        ?meal pm:HasIngredient ?ingredient .
20    }
```

```

20     ?ingredient a pm:AnimalBasedIngredient .
21     BIND("Contains meat/fish" AS ?reason)
22 }
23 }

```

Aggregates reasons for exclusion, making rule output interpretable.

7.7.6 6. List All Meals and Their Caloric Values

```

1 SELECT ?meal (STR(?calories) AS ?calorieValue)
2 WHERE {
3     ?meal a pm:Meal ;
4           pm:Calories ?calories .
5 }

```

Retrieves caloric values for all meals.

7.7.7 7. Guest Profiles – Dietary and Calorie Preferences

```

1 SELECT ?guest ?diet ?calPref
2 WHERE {
3     ?guest a co:Guest .
4     OPTIONAL { ?guest co:hasDietType ?diet . }
5     OPTIONAL { ?guest co:hasCaloriePreference ?calPref . }
6 }

```

Lists guests with their dietary and caloric preferences.

7.7.8 8. Meals with Allergens and Their Ingredients

```

1 SELECT ?meal ?allergen ?ingredient
2 WHERE {
3     ?meal pm:ContainsAllergen ?allergen .
4     ?meal pm:HasIngredient ?ingredient .
5 }

```

Displays allergen-containing meals and their ingredients.

8 Ontology-Aware Process Modeling

8.1 Agile, Ontology-Based Metamodeling (AOAME)

This chapter explores the innovative approach of **Agile, Ontology-Based Metamodeling (AOAME)**, a methodology specifically designed to increase the adaptability, precision, and long-term relevance of modeling languages, particularly in complex and dynamic domains such as business processes, knowledge systems, and personalized services.

AOAME combines the iterative, flexible nature of agile development with the semantic depth of ontology-driven frameworks. The result is a metamodeling environment that evolves gradually, in short, controlled development cycles, while maintaining formal connections to domain knowledge represented within ontologies. This ensures that models remain not only technically correct but also semantically meaningful and aligned with real-world concepts.

In practice, AOAME allows the continuous refinement and extension of modeling languages by introducing new elements or relationships derived from shared domain ontologies. It fosters collaboration between technical experts and domain specialists, making the models both adaptable to changing requirements and robust from a knowledge representation perspective.

In the context of this project, AOAME was applied to extend the capabilities of BPMN 2.0, incorporating concepts from the restaurant ontology developed earlier. This enabled the creation of intelligent, ontology-aware process models capable of interacting with a personalized menu recommendation system.

8.2 Business Process Modeling with BPMN 2.0

8.2.1 What is BPMN 2.0

Business Process Model and Notation (BPMN) version 2.0 is an international standard developed by the Object Management Group (OMG) for modeling and visualizing business processes. BPMN 2.0 provides a rich graphical language designed to describe workflows, decisions, and process structures in a way that is understandable to both technical teams and business stakeholders.

This standard includes symbols for representing tasks, events, decision gateways, and the flow of activities within a process. In addition to its visual representation, BPMN 2.0 defines formal semantics and provides a machine-readable XML format that allows models to be executed or integrated with automated systems.

Although BPMN 2.0 is highly effective for representing general business processes, it often lacks the domain-specific semantics needed for specialized applications. In this project, it was necessary to extend BPMN 2.0 with new modeling elements linked to the restaurant ontology, such as tasks for suggesting meals based on guest profiles or processes that incorporate allergen filtering. This was made possible through the AOAME approach, which ensures the extended models remain consistent with the underlying knowledge base while also being adaptable to future changes in requirements.

8.2.2 Our BPMN Model

Our BPMN model represents the interaction between the guest and the restaurant system, structured across two distinct lanes: the **Guest Lane** and the **Restaurant System Lane**. The process begins when the guest initiates interaction by scanning the QR code provided at the restaurant table. This action triggers the next step, where the guest enters their personal preferences, including dietary choices, allergy information, and calorie-conscious indications.

Following this, a message is generated and sent from the Guest Lane to the Restaurant System Lane, specifically labelled **Send Preferences to the System**. On the system side, the process

continues with the task **Receive Preferences**, which records the guest’s input data and makes it available for subsequent processing.

At this point, the most critical part of the process is executed through an **Extended Task** called **Create Personalized Menu for the Guest**. This task differs from standard BPMN elements as it has been semantically extended following the principles of Agile Ontology-Based Metamodeling. The extended task introduces additional properties required to interface with the knowledge-based recommendation system. Specifically, this task includes the following properties: **DietType**, **CaloriePreference**, and **Allergen**. Each of these properties is structured as a string data type, containing values directly derived from the guest’s selections.

These properties are not merely for visualization but serve a functional purpose in the system’s interaction with **Jena Fuseki**, the triple store and SPARQL endpoint. The values captured within these properties are read by Jena and dynamically injected into SPARQL queries. These queries retrieve and filter meals from the knowledge graph, based on the guest’s declared preferences and restrictions, ensuring that the menu presented is fully personalized and consistent with both the ontology structure and the guest’s individual requirements.

Once the personalized menu has been generated, the process concludes with the **Send Final Menu to the Guest** task within the Restaurant System Lane, followed by the **Receive Custom Menu** task in the Guest Lane, where the guest receives their filtered, tailored menu on their device. At this point, the BPMN process terminates.

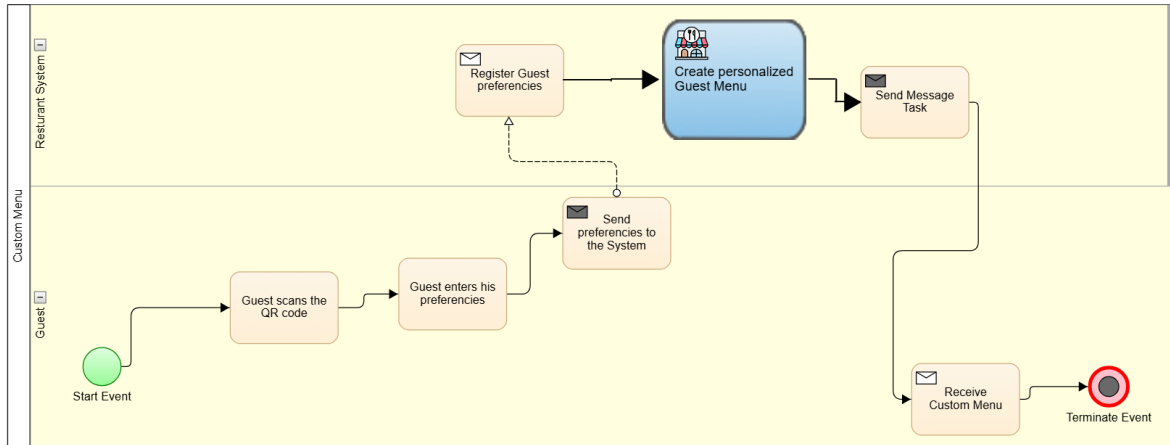


Figure 9: BPMN model using AOAME for guest preference integration

During development, we encountered technical limitations with the AOAME. To overcome this, we designed an additional version of the process model using **Camunda**, a platform that provides greater flexibility in process modeling and allows for more realistic, executable diagrams. However, Camunda does not natively support the extension of BPMN elements with custom ontology-linked properties, which is why this second version, while structurally complete and more accurate in representing real-world interactions, lacks the semantic enrichments present in the AOAME-based model.

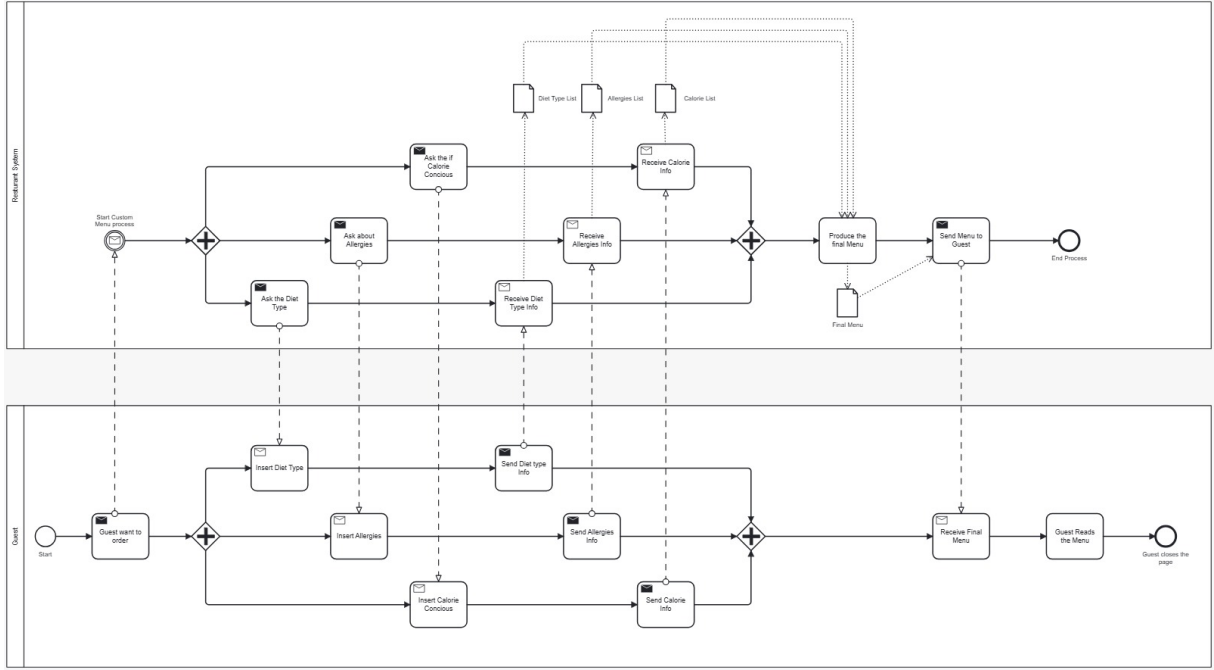


Figure 10: BPMN model using Camunda

8.3 Jena Fuseki

Jena Fuseki is an open-source SPARQL server and triple store developed as part of the **Apache Jena** framework, which is widely used for building semantic web and linked data applications.

Fuseki provides a ready-to-use server for storing, querying, and managing **RDF (Resource Description Framework)** datasets. RDF is the standard data model for representing structured knowledge in ontologies and knowledge graphs, where information is stored as subject-predicate-object triples.

The primary function of Jena Fuseki is to expose RDF data via a **SPARQL endpoint**, meaning it allows external systems, applications, or users to send SPARQL queries over HTTP and retrieve results. SPARQL (SPARQL Protocol and RDF Query Language) is the standard query language for querying and manipulating RDF data.

8.3.1 Query in Jena Fuseki

On the **Jena Fuseki** server, we implemented a SPARQL query designed to dynamically retrieve a personalized menu from the ontology. This ontology, structured according to the principles of AOAME, was semantically extended to include the specific classes and properties required to manage guest preferences within the BPMN process.


```

1 PREFIX mo: <http://purl.org/ontology/mo/>
2 PREFIX pm: <http://example.org/personalized-menu#>
3 PREFIX recomm: <http://www.semanticweb.org/fabio/ontologies/2025/5/PerzonalizedMenu#>
4 PREFIX co: <http://www.co-ode.org/ontologies/ont.owl#>
5 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
6 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
7 PREFIX bpaas: <http://ikm-group.ch/archimeo/bpaas#>
8 PREFIX mod: <http://fhnw.ch/modelingEnvironment/ModelOntology#>
9 PREFIX lo: <http://fhnw.ch/modelingEnvironment/LanguageOntology#>
10
11 SELECT DISTINCT ?mealName WHERE {
12   mod:CreatepersonalizedGuestMenu_a9a5020c-c09b-410b-8fba-edf40d581c1d
13     lo:Allergen ?allergenLit ;
14     lo:DietType ?dietLit ;
15     lo:CaloriePreference ?cprefLit .
16
17   # Trasformazione in IRI
18   BIND(IRI(CONCAT("http://example.org/personalized-menu#", STR(?allergenLit))) AS ?allergen)
19   BIND(IRI(CONCAT("http://www.co-ode.org/ontologies/ont.owl#", STR(?dietLit))) AS ?diet)
20   BIND(IRI(CONCAT("http://www.co-ode.org/ontologies/ont.owl#", STR(?cprefLit))) AS ?cpref)
21
22   ?meal a pm:Meal ;
23         pm:MealName ?mealName ;
24         pm:Calories ?cal .
25
26   # Filtro 1: Allergen
27   FILTER NOT EXISTS {
28     ?meal pm:ContainsAllergen ?allergen .
29   }
30
31   # Filtro 2: Dieta
32   FILTER NOT EXISTS {
33     ?meal pm:HasIngredient ?i .
34     ?i a pm:AnimalBasedIngredient .
35     FILTER(?diet = co:Vegetarian || ?diet = co:Vegan)
36   }
37
38   # Filtro 3: Calorie
39   FILTER(
40     (?cpref = co:CalorieConscious && ?cal <= 600) || (?cpref = co:NotCalorieConscious)
41   )
42 }

```

Figure 11: Jena Fuseki query

The query exploits these extended classes, which were carefully designed to serve as a bridge between the business process model and the knowledge graph. Thanks to this design, guest preferences such as dietary restrictions, allergens, and calorie-conscious indications are no longer statically defined within the query itself. Instead, these values are passed from the BPMN model, recorded within instances of the extended class, and automatically retrieved by the SPARQL query at runtime. This separation of data and query logic significantly improves scalability, modularity, and maintainability of the system.

Model element attributes

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

Instantiation Type: Instance

Relation	Value	Actions
DietType	<input type="text" value="Vegetarian"/>	<button>Remove</button>
CaloriePreference	<input type="text" value="CalorieConscious"/>	<button>Remove</button>
Allergen	<input type="text" value="Dairy"/>	<button>Remove</button>

▼ Add Relation

Save **Close**

Figure 12: Semantic annotation of the task **Create personalized Guest Menu** using AOAME. Parameters like **DietType**, **CaloriePreference**, and **Allergen** are attached to enable semantic reasoning.

This approach ensures seamless communication between the business process and the knowledge graph, allowing the system to generate a fully personalized list of suitable meals based on the real-time preferences captured during the BPMN process execution.

The results of the query execution are presented below, showcasing the personalized menu generated based on the guest preferences previously defined. These outcomes validate the seamless interaction between the BPMN process, its AOAME extensions, and the Jena Fuseki server, demonstrating the system's capability to deliver precise, ontology-driven recommendations aligned with both the technical goals and the practical objectives of the project.

	mealName
1	Acqua Naturale / Frizzante
2	Limoncello
3	Aperol Spritz
4	Prosecco
5	Aranciata (Orange Soda)
6	Pasta al Pomodoro
7	Zuppa di Farro e Legumi
8	Caffè Espresso
9	Chinotto
10	Succo di Frutta (Fruit Juice)
11	Marinara
12	Marinara no Gluten
13	Tofu alla Mediterranea
14	Vino Rosso della Casa
15	Vino Bianco della Casa

Showing 1 to 15 of 15 entries

Figure 13: Jena Fuseki results

9 Conclusion Claudio

The *Personalized Menu* project demonstrated the practical benefits of integrating knowledge-based technologies, ontologies, and formal process modeling to develop an intelligent and adaptable meal recommendation system tailored to individual guest preferences. Through the combined use of Decision Tables, Prolog logic, and an ontology-driven knowledge graph, supported by tools such as **Trisotech Decision Modeler**, **SWISH Prolog**, **Protégé**, **AOAME**, and **Jena Fuseki**, the project successfully created a system capable of filtering and recommending meals based on dietary needs, allergies, and calorie-conscious preferences.

A key achievement of the project was the semantic extension of **BPMN 2.0** using **Agile, Ontology-Based Metamodeling (AOAME)**. This allowed process models to directly interact with the ontology, ensuring that guest preferences captured during the process flow were seamlessly passed to the knowledge base for intelligent filtering. The integration with **Jena Fuseki** further enabled real-time, query-based interaction with the ontology, resulting in accurate, context-aware menu suggestions.

The system was rigorously tested with both realistic menu datasets and exhaustive test cases designed to simulate all possible combinations of dietary profiles and allergen sensitivities. Results consistently confirmed that the system produces reliable, logically sound recommendations aligned with user expectations.

9.1 Advantages and Disadvantages of the Knowledge-Based Solutions

The project explored three distinct knowledge-based approaches, each offering unique strengths and limitations:

9.1.1 Decision Tables with Trisotech Decision Modeler

The decision tables provided a clear, structured way to model filtering logic based on user preferences. By externalizing the menu data via XML files, the system achieved flexibility and scalability, as the dataset can be updated without modifying the decision logic itself. Additionally, the use of FEEL (Friendly Enough Expression Language) expressions allowed for more complex, realistic filtering behavior. However, a key limitation emerged: the standard decision table structure proved insufficient for representing the full complexity of menu filtering, particularly when dealing with dynamic lists and nested properties. This necessitated an approach based more on custom scripting than on traditional decision tables, which, while more powerful, sacrificed some of the simplicity and visual clarity typically associated with decision modeling.

9.1.2 Prolog Logic Programming

The Prolog-based solution showcased the power of declarative logic for knowledge representation and reasoning. The use of facts and rules allowed for precise, formal definitions of meal properties, dietary restrictions, and allergen relationships. The recursive, logic-driven structure made it possible to perform complex filtering with relatively concise code. Prolog's natural support for rule chaining and backtracking offered flexibility in handling various combinations of preferences. Nevertheless, Prolog's syntax and logic programming paradigm may present a steep learning curve for stakeholders unfamiliar with formal logic. Additionally, integration with user interfaces or other systems often requires additional development effort compared to more visually oriented tools.

9.1.3 Ontology and Knowledge Graph with Jena Fuseki

The ontology-driven solution, implemented with Protégé and accessed through Jena Fuseki, provided the most semantically rich and scalable approach. By formally modeling meals, ingredients,

allergens, and guest profiles within a shared knowledge graph, the system ensured consistency, extensibility, and alignment with semantic web standards. The ability to execute dynamic SPARQL queries based on real-time BPMN inputs allowed for accurate, ontology-consistent filtering. The main disadvantage of this approach lies in its technical complexity. Designing and maintaining ontologies requires specialized knowledge, and working with SPARQL queries can be less intuitive for developers unfamiliar with semantic technologies. Furthermore, performance considerations arise as the dataset grows, making optimization necessary for large-scale deployments.

10 Conclusion Fabio

This project allowed me to consolidate and significantly expand my knowledge in semantic technologies and knowledge-based systems. While I had previously worked with **Protégé** in another context, this experience gave me the opportunity to explore its more advanced functionalities, such as SWRL rule development, SHACL shape validation, and tighter integration with SPARQL querying—capabilities that proved essential for constructing a consistent and intelligent ontology.

10.1 Semantic Ontology Engineering: From Modeling to Inference

A central focus of my work was the design of the ontology: I defined the core conceptual structure, including the class hierarchy and their semantic relationships. The ontology was designed to be modular, expressive, and reasoning-ready. Instead of modeling all possible meal classifications explicitly (as I had done in a previous project using verbose SPARQL queries and multiple intermediate classes), I adopted a more semantic and compact strategy: general-purpose SWRL rules infer the `notRecommendedFor` property when a meal violates a guest’s preferences (e.g., allergies, dietary restrictions, or calorie concerns). This allowed the SPARQL logic to remain clean and declarative.

For example, the SPARQL queries in this system are greatly simplified thanks to the reasoning layer: rather than filtering based on static flags or manually binding parameters, a simple `FILTER NOT EXISTS` is sufficient to exclude disrecommended meals. This not only improves maintainability but also separates domain logic from query structure—an important design principle in knowledge-based systems.

The use of SHACL further strengthened the ontology by enforcing structural and semantic constraints across multiple classes. I defined a set of SHACL shapes to ensure data consistency, including rules that required each `Meal` to declare its ingredients and calorie value, enforced datatype correctness for numerical and string fields, and validated the correct use of object properties such as dietary tags and allergen links.

10.2 Integrating Semantic Models with Business Processes

Beyond ontology modeling, I also actively contributed to the development of the second task. Working in close collaboration with my teammate Claudio, we extended the BPMN 2.0 process using the **Agile, Ontology-Based Metamodeling (AOAME)** methodology. Together, we designed and annotated extended BPMN elements with semantic properties such as `DietType`, `CaloriePreference`, and `Allergen`, establishing a direct connection between the process model and the underlying ontology.

We then reused the knowledge base in a dynamic context through **Jena Fuseki**, adapting and rewriting the SPARQL queries to accept real-time parameters coming from the BPMN process (e.g., guest profile ID). This required careful restructuring to preserve query correctness while enabling runtime customization of the menu recommendation. This phase of the project was particularly stimulating, both from a technical and collaborative perspective, as it combined multiple tools and required a coherent integration across disciplines.

10.3 A Comparative Perspective: Toward a More Elegant Semantic Architecture

In comparison with my previous ontology-based project, the solution developed here is both more elegant and more powerful. Rather than encoding logic redundantly in the query itself or in class hierarchies, the ontology handles inference natively using SWRL rules and inverse properties like `recomm:notRecommendedFor`. This resulted in a cleaner, semantically-grounded architecture that better separates concerns and is easier to maintain.

Overall, the project was a highly enriching experience. It gave me the opportunity to refine my skills in ontology modeling, reasoning, and validation, while also working across different tools and methodologies in a coherent and collaborative development effort. Most importantly, it validated the effectiveness of combining process modeling and semantic technologies to address real-world personalization challenges in a structured and scalable way.

11 Final Considerations

The *Personalized Menu* project demonstrates how knowledge-based systems, process modeling, and semantic technologies can be effectively combined to solve practical challenges in personalized service delivery. By integrating Decision Tables, Prolog logic, Ontology-based reasoning, and BPMN process models extended through AOAME, the system provides a comprehensive, flexible, and scalable solution for tailoring restaurant menus to individual guest needs.

Each technology contributed specific strengths to the overall system. Decision modeling introduced clarity and structure, Prolog enabled formal reasoning, while the ontology and knowledge graph ensured semantic consistency and extensibility. The use of Jena Fuseki to connect the knowledge graph with real-time process execution further reinforced the practical viability of the approach.

Nevertheless, the project also highlighted certain technical and conceptual trade-offs. Increased complexity, the need for specialized knowledge in logic and semantic technologies, and tool limitations in fully integrating BPMN extensions represent areas for improvement. Despite these challenges, the implemented system reliably produced personalized, filtered menus aligned with guest preferences and dietary requirements.

Ultimately, the project confirms the potential of combining process models, knowledge bases, and semantic technologies to enhance user experiences in digital environments. The methodologies applied here are not only applicable to restaurants but also adaptable to other industries where personalization, knowledge management, and intelligent decision support are required.

This work lays a solid foundation for further development, including the possibility of integrating machine learning, automating data updates, and extending the knowledge graph to cover a broader range of dietary needs, cultural variations, or health considerations.