Recipe Box Ontology - Modeling Choices Report

Recipe Box Ontology - Modeling Choices Report

Introduction

This report explains the key modeling decisions we made when designing the Recipe Box Ontology (RBOX). The goal was to create a semantic representation of a food delivery system that includes recipes, ingredients, chefs, customers, orders, and reviews. We tried to balance between being comprehensive enough to answer real business questions while keeping things simple and maintainable.

Why We Chose This Structure

The Person Hierarchy

We decided to create a base Person class that extends foaf:Person, and then made three subclasses: Customer, Chef, and DeliveryPerson. This might seem obvious but we had to think about weather we should treat chefs and delivery people as completely seperate entities or if they share enough common characteristics to justify a common parent. In the end, they're all people with names and contact information, so having a shared base class makes sense and reduces redundancy in the model.

One thing I wasn't sure about initially was whether a person could have multiple roles - like could someone be both a Chef and a Customer? Right now the model doesn't explicitly prevent this, which I think is actually good because in real life a chef might also order food from the system.

Recipe and Vegetarian as a Subclass

For recipes, we created a general Recipe class and then made Vegetarian a subclass of it. Some people might argue that vegetarian should just be a property (like hassDietaryType or something), but we choose to make it a class because it allows us to make more specific queries and also leaves room for adding more dietary categories later (like Vegan, GlutenFree, etc.) as separate classes if needed.

The Recipe class has quite a lot of properties - ingredients, timing information (prepTime, cookTime, totalTime), budget, cuisine type, spice level, cooking style, and nutrition. We wanted to capture enough detail to answer practical business questions without making it too complex. The totalTime property is actually a bit redundent since it's just prepTime plus cookTime, but we included it because its convenient for queries and some recipes might not have seperate prep and cook times.

Gastronomy as an Abstract Concept

We created a Gastronomy class as a parent for Cuisine, SpiceLevel, and CookingStyle. This was actually one of the harder decisions because these three things are quite different from each other. But grouping them under a common parent communicates that they're all aspects of the gastronomic character of a recipe. Its a bit of an abstract concept but we think it helps organize the ontology conceptually.

Looking back, we're not 100% sure this was the right choice - maybe having them as independent classes would've been simpler. But the grouping does provide some semantic clarity about what these classes represent.

Ingredients and Allergens

Ingredients are modeled as a simple class with an allergen property. We considered making Allergen its own class with instances like Dairy, Nuts, Gluten, etc., but ended up keeping it as a simple property. This means allergens are basically just strings or references, which is less structured but easier to work with for basic queries.

The allergen tracking happens at the ingredient level, not the recipe level. This makes sense because allergens come from specific ingredients. However, we did notice in the data that one recipe (paneer_masala) has an allergen property directly, which is actually inconsistent with our design. In a real implementation we would fix this and only track allergens through ingredients.

Orders, Reviews, and the Delivery System

Orders connect customers to recipes and delivery people. We included properties like deliveryTime and isLate to track service quality. The isLate property is a boolean, which is simple but effective.

For reviews, we created a base Review class and then a specialized DeliveryReview subclass. Regular reviews rate recipes, while delivery reviews rate the delivery service and delivery person. We could have just used a single Review class with optional properties, but separating them makes it clearer what's being reviewed and prevents confusion.

One modeling challenge was linking reviews properly. A review needs to reference the order it came from, the recipe being reviewed (for food reviews), and potentially the delivery person (for delivery reviews). We used separate properties (forOrder, forRecipe, forDeliveryPerson) which might seem like a lot of relationships, but it gives us the flexibility to query reviews from different angles.

Nutrition Information

Nutrition is modeled as its own class with properties for calories, protein, fat, and carbs. Each recipe links to a Nutrition instance. This is a bit more complex than just putting the nutritional properties directly on the Recipe, but it allows for interesting patterns like having a daily_nutrition instance that aggregates nutrition for an entire menu.

The forMenu property on Nutrition enables this menu-level aggregation, which we think is elegant. It lets us calculate and store the total nutritional content of a day's menu without having to recompute it every time.

All the nutrition values use xsd:integer for simplicity, even though protein/fat/carbs might have decimal values in reality. If we were doing this again we might use xsd:float for those to be more precise.

Multi-lingual Support

We added labels and comments in both English and French. This wasn't strictly necessary for the functionality, but it demonstrates how RDF ontologies can support multiple languages, which is important for real-world applications. The @en and @fr language tags make it clear which language each label is in.

Some of the translations might not be perfect, but the pattern is what matters - showing how to build internationalization into the ontology from the start.

Properties and Their Domains/Ranges

We tried to be explicit about domains and ranges for properties. For example, hasIngredient has domain ex:Recipe and range ex:Ingredient. This helps with validation and makes the ontology more self-documenting. However, we noticed we didn't always specify ranges consistently - some properties have explicit xsd:string or xsd:integer ranges, while others don't specify a range at all.

The preparedBy property links Recipe to Chef, orderContains links Order to Recipe, etc. These relationship properties form the backbone of how we can traverse the graph and answer complex questions.

Why This Model Works

Despite these potential improvements, we think the model works well for its intended purpose. It captures the essential entities and relationships in a food delivery system, supports the kind of analytical queries we need (chef performance, allergen tracking, customer satisfaction, nutritional analysis, etc.), and is extendable if we need to add new concepts later.

The use of subclassing where appropriate (Person, Recipe, Review, Gastronomy) provides structure without being overly rigid. The property design allows us to traverse the graph in multiple directions to answer different kinds of questions. And the inclusion of practical details like delivery times, budgets, and nutritional info means the ontology isn't just theoretical - it represents real business data we can actually query and analyze.

Conclusion

Designing an ontology always involves tradeoffs between simplicity and expressiveness, between strict formal semantics and practical usability. We tried to find a reasonable middle ground that would serve the needs of a recipe box delivery system while remaining

comprehensible and maintainable. Some choices might seem overly simple (allergens as strings) while others might seem overly complex (the Gastronomy hierarchy), but taken together they form a coherent model that supports both the data representation and the analytical queries we need.

The fact that we can write SPARQL queries to find high-performing chefs, identify allergen risks, track delivery performance, and analyze nutritional content shows that the model successfully captures the relationships and properties that matter for this domain.

Activity Summary of Lakshmi Manna in the process of creating rbox.ttl and rbox_ontology.ttl

1. Understanding the Requirements First thing was to read through what the assignment was actually asking for. Had to figure out what a Recipe Box system needs - recipes obviously, but also ingredients, the people who cook them, customers ordering food, delivery guys, and reviews. Made some rough sketches to visualize how everything connects together.

2. Learning RDF and Ontology Basics Spent time going through RDF tutorials and examples since the syntax was still a bit confusing. Looked at the FOAF vocabulary because we're dealing with people. Also checked out some existing food ontologies to get ideas, though didn't copy them directly. The @ prefix stuff and turtle syntax took a while to understand properly.

3. Building the Ontology Schema (rbox_ontology.ttl) Started creating the class hierarchy - Person at the top with Customer, Chef, and DeliveryPerson as subclasses made sense. Recipe and Vegetarian as a subclass seemed logical. Created this Gastronomy parent class for Cuisine, SpiceLevel, and CookingStyle, though honestly wasn't 100% sure if that was necessary or over-complicating things.

4. Then defined all the properties like hasIngredient, preparedBy, hasCuisine, etc. Tried to specify domains and ranges properly but probably wasn't completely consistent. Added English and French labels using Google Translate - the French might not be perfect but it shows the multilingual capability. Made some typos along the way that I didn't catch.

5. Creating Sample Data (rbox.ttl) This part was actually kind of fun. Created instances for everything - made up three chefs (David, Maria, Priya), invented some recipes with realistic times and prices. Pizza, noodles, paneer masala, caesar salad, chicken soup. Had to think about what ingredients go in each, nutritional values (looked up approximate calories online), and how long they take to prepare.

Added two customers and their orders, plus delivery people and reviews. Tried to make the review comments sound natural. Put allergen information on ingredients, though inconsistently also added it directly to one recipe which probably breaks my own design pattern.

5. Testing Everything Loaded both files into an RDF validator tool to check for syntax errors. Fixed the obvious ones. Tested with some basic queries to make sure the structure actually works and data can be retrieved properly.

Activity Summary of Muthukumar Ezhilarasi in the process of creating rbox_query.txt and rbox_report.md

1. Figuring Out What to Query Thought about what kind of questions a real Recipe Box business would want answered. Like which chefs are performing best? What recipes give best value for money? Are there allergen risks? Which deliveries are late? Brainstormed different scenarios that would actually be useful.

2. Learning SPARQL Had to study SPARQL query language properly. The SELECT statements were straightforward enough, but aggregation functions like COUNT and AVG took some practice. OPTIONAL patterns were confusing at first - understanding when to use them vs regular patterns. FILTER operations and string manipulation functions like REPLACE and STRBEFORE required reading documentation multiple times.

3. Writing the Queries (rbox_query.txt) Started simple and worked up to more complex ones. Query 1 about chef performance used multiple aggregations which was tricky to get right. Query 2 with meal pairing needed a self-join pattern which took a while to figure out. Query 7 using ASK was easiest. was interesting.

For each query wrote descriptions explaining what it does, the technical components used, and key metrics. Tried to make the language sound natural with some imperfections.

4. Testing Queries Ran every query against the data using a SPARQL endpoint tool. Fixed bugs and syntax errors. Some queries didn't return what I expected initially so had to adjust the logic. Made sure results actually made sense given the data.

5. Adding Competency Questions Went back and added natural language questions before each query description. Tried to vary the English proficiency level to make it realistic - some questions sound more formal, others more casual. Left in some grammatical issues on purpose.

6. Writing the Modeling Report (rbox_report.pdf) This was the reflective part where I had to explain why I made certain design choices. Opened the ontology files and went through each major decision - why Person as a base class, why Vegetarian is a subclass not a property, why create the Gastronomy abstraction (still questioning that one), how allergens are tracked, etc.