BS Portfolio 01 MealPlanner

Merlin Willner [5124045]
Luis Schneider [XXX]
Hannes Ebert [5124082]
Raphael Geiling [5124059]

October 27, 2025

1 Project Overview

Elevator Pitch: Food prices for similar products vary dramatically, often by several multiples. MealPlanner helps people reach their fitness and nutrition goals on a budget by showing them which foods and dishes give them the best value for their money based on what they want to achieve, whether that's losing weight, building muscle, or both.

Persona 1 – Budget-Conscious Student

Problem: He weighs 90 kg and wants to increase his muscle mass. However, as a student, he has a tight budget for food. What is the most cost-effective way to get enough calories and protein without breaking the bank?
Solution: MealPlanner allows users to filter foods by:

- Price per 100 g of protein
- Price per 1000 calories

This helps find affordable options that meet his nutritional needs.

Persona 2 – Time-Pressed Professional

Problem: She wants to lose weight but has a busy schedule and limited time for meal prep. What are the best meal options that are both healthy and quick to prepare?
Solution: MealPlanner enables her to create and search for dishes based on:

- Calories per serving
- Caloric density (calories per 100 g)
- Preparation time

This allows her to find meals that fit her weight loss goals and time constraints.

What We Built. Our system manages four main components: *FoodItem*, *Dish*, *User*, and *ShoppingCart*. Users can add new *FoodItem* entries and create personalized *Dish* combinations from them. Once a *Dish* exists, the system automatically calculates nutritional values and costs. Users can then filter dishes by their nutritional value, cost, or the required time to prepare them. Once users find a suitable option, they can add all ingredients (with the corresponding quantities) directly to their *ShoppingCart* for convenient purchasing.

Scope Limitations. The first increment explicitly excludes allergen/micronutrient modeling and differentiation between coach/client roles – all users have the same capabilities to create food items and dishes as well as filter and purchase them. Unit differentiation (pieces, liters vs. grams) is also excluded to keep the deliverable focused.

2 Domain Model

The following domain model describes the main objects of the system. It shows important entities, their attributes, and how they are connected. The system focuses on users, food items, dishes, and a shopping cart for buying ingredients.

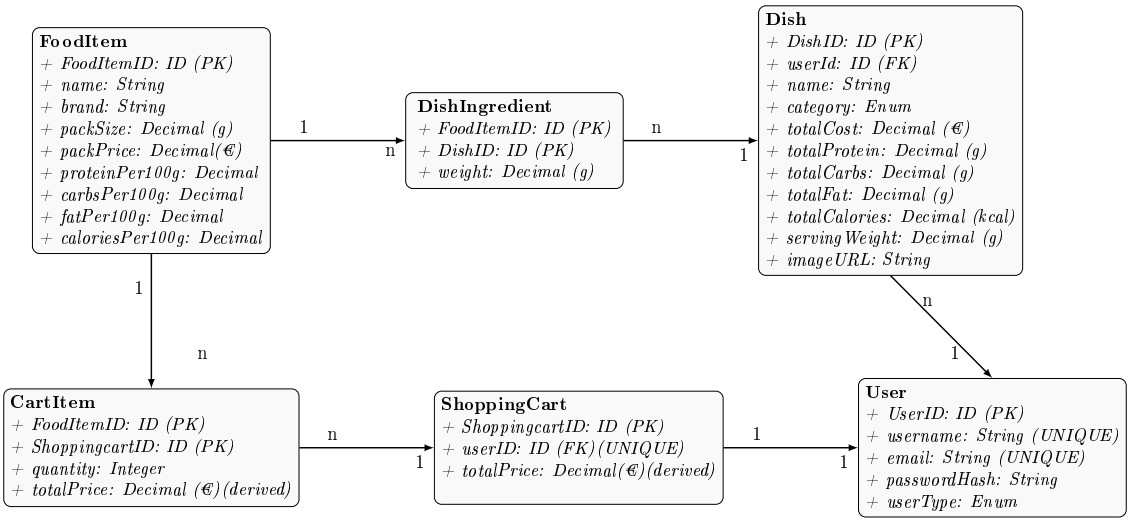

Figure 1: UML class diagram of the Meal Planner domain.

Main Entities

FoodItem represents a food product in a store. It includes nutritional information such as protein, carbohydrates, fat and calories per 100g, and also the pack size and price. A food item can be used in many dishes and can also appear many times in shopping carts.

DishIngredient connects a *Dish* with a *FoodItem*. It stores how many grams of the food item are used in the dish. This creates a many-to-many relationship between dishes and food items.

Dish is a recipe created by a user. The *category* is an enum that stores if the dish is breakfast, lunch, dinner, dessert, snack or something else. It also saves the *total cost* in euros, *total protein*, *total carbohydrates*, *total fat*, *total calories*, and the *serving weight* in grams. A dish can also have an optional *image link* to upload a photo of the Dish.

CartItem connects a *FoodItem* with a *ShoppingCart*. It stores the *quantity* of the food item. The *total price* for the item is calculated from the quantity.

ShoppingCart belongs to one user. It can contain many *CartItem* objects. The *total price* of the shopping cart is calculated from all items inside it.

User represents a registered person in the system. Each user has a unique *username* and *email*. The *password* is saved as a hash for security. The *userType* is an enum that stores the user's goal, such as building muscle, losing weight, gaining weight or living a healthier lifestyle. A user can create many dishes, but each user has only one shopping cart.

Rules and Constraints

- Each username and email must be unique.
- A user can only have one shopping cart at the same time.
- Nutritional values and prices cannot be negative.
- Quantity in a cart and weight in a dish must be greater than 0.
- Total calories, total cost, and other totals are calculated automatically.

Example Records

Example FoodItem

```
FoodItemID = 101
name = "Oats"
brand = "BioFarm"
packSize = 1000 g
packPrice = 2.79 €
proteinPer100g = 12 g
carbsPer100g = 60 g
fatPer100g = 7 g
caloriesPer100g = 370 kcal
```

Example Dish

```
DishID = 22
userID = 5
name = "Protein Oatmeal"
category = Breakfast
servingWeight = 350 g
totalCalories = 520 kcal
totalProtein = 32 g
totalCost = 1.10 €
```

These examples show how real data can appear in the system. The domain model gives a clear and structured overview of users, food items, dishes, and shopping carts. It supports the calculation of nutrition values and costs, the management of ingredients, and the planning of meals and shopping. In this way, the model helps to create an organized and efficient system for healthy eating and meal preparation.

3 Use Cases

The backend covers six core use cases. Each use case follows a clear structure with preconditions, a main flow, alternate/failure flows, postconditions, and acceptance criteria.

UC01 – Register Food Item (Must)

Primary Actor: User
Goal (User Story): "As a user, I want to register an ingredient with correct nutrition and pricing so later calculations are reliable."
Preconditions: The name is unique; pack size and price are positive; macro values per 100 g are within sensible ranges; brand is optional.
Main Success Scenario:

1. The actor submits `POST /food-items` with name, optional brand, pack size (g), pack price, and protein/carbs/fat/calories per 100 g.
2. The service validates values, rounds to two decimals, and computes helper metrics (e.g., price per 100 g).
3. The database stores the `FoodItem` with timestamps.
4. The API returns `201 Created` with the id and the saved record.

Alternate / Failure Flows:

1F. Duplicate name → `409 Conflict` with instructions to choose a distinct name.

2F. Out-of-range or negative values → `400 Bad Request` with details.

Postconditions: The `FoodItem` is stored in a standard format and can be used to compose dishes.
Acceptance Criteria:

- Numeric fields are stored with two decimals; invalid values are rejected with clear messages.
- Derived metrics (e.g., price per 100 g) are correct to two decimals.

UC02 – Compose Dish (Must)

Primary Actor: Nutrition planner
Goal (User Story): "As a planner, I want to create a dish from precise ingredient weights so totals are computed automatically."
Preconditions: At least one `FoodItem` exists; each ingredient grams value is > 0; each `FoodItem` appears at most once in the dish.
Main Success Scenario:

1. The actor calls `POST /dishes` with name, servings, and an ingredients array of {`foodItemId`, `grams`}.
2. The application service loads the referenced `FoodItem`s and computes total cost and macronutrients; per-serving values use `servingGrams`.
3. The transaction persists `Dish` and `DishIngredient` entities atomically.
4. The API returns `201 Created` with computed totals and a version number for concurrency.

Alternate / Failure Flows:

1F. Unknown `foodItemId` → `404 Not Found`.

2F. Duplicate `foodItemId` in the payload → `400 Bad Request` with guidance to merge grams.

3F. Ingredient grams $\leq 0 \rightarrow$ `400 Bad Request` with validation details.

Postconditions: The dish is stored with correct totals and can be listed and adjusted.
Acceptance Criteria:

- Calculated totals equal the sum of ingredient contributions within rounding tolerance.
- Saved and fetched representations match, including totals and per-serving values.

UC03 – Adjust Dish Composition (Should)

Primary Actor: Nutrition planner
Goal (User Story): "As a planner, I want to adjust ingredient weights so that cost and nutrition metrics remain accurate over time."
Preconditions: The dish exists and currently references at least one ingredient.
Main Success Scenario:

1. The actor uses sub-resources to modify composition:
   - Add: `POST /dishes/{dishId}/ingredients` with {`foodItemId`, `grams`}.
   - Update: `PATCH /dishes/{dishId}/ingredients/{dishIngredientId}` with {`grams`}.
   - Remove: `DELETE /dishes/{dishId}/ingredients/{dishIngredientId}`.
2. The domain validates all changes and recomputes totals, per-serving metrics, and derived rankings.
3. The persistence layer commits in a single transaction and updates timestamps.
4. The API returns the refreshed `Dish` with an incremented version.

Alternate / Failure Flows:

1F. Concurrent modification $\rightarrow$ `409 Conflict` with retry guidance.

2F. Removing all ingredients $\rightarrow$ `400 Bad Request` with "Dish must contain at least one ingredient."

Postconditions: The dish reflects updated totals and remains consistent for listing and ranking.
Acceptance Criteria:

- Per-serving calculations update consistently after each modification.
- Versioning prevents overwrites and clearly signals conflicts.

UC04 – Add Dish to Shopping Cart (Should)

Primary Actor: Shopper preparing groceries
Goal (User Story): "As a shopper, I want to add a whole dish to my shopping cart so that all its ingredients are added in the correct quantities automatically."
Preconditions: The target `Dish` exists; an active shopping cart exists (or is created implicitly); `servingsMultiplier` is a positive integer.
Main Success Scenario:

1. The actor calls `POST /shopping-carts/{cartId}/items/from-dish` with {`dishId`, optional `servingsMultiplier`=1}.

2. The application service loads the dish and its `DishIngredients`.

3. For each ingredient, the service creates or increments a `CartItem` for the `foodItemId` with grams = `ingredient.grams` × `servingsMultiplier`.

4. The transaction persists all item changes and returns the updated cart state.

Alternate / Failure Flows:

1F. Unknown `dishId` or `cartId` → `404 Not Found`.

2F. Non-positive `servingsMultiplier` → `400 Bad Request` with validation details.

3F. Concurrent write to the same cart → `409 Conflict`; ask the user to retry.

Postconditions: The shopping cart contains one item per `FoodItem` in the dish; grams are merged into existing items instead of duplicating lines.
Acceptance Criteria:

- Adding a dish creates or increments per-`FoodItem` items by `ingredient.grams`×`servingsMultiplier`.
- Re-adding the same dish with `servings = n` increases the same items cumulatively; no duplicate rows are created.
- When two dishes share a `FoodItem`, the cart reflects the summed grams across both additions.

UC05 – Filter and Rank Food Items (Must)

Primary Actor: Shopper or nutrition planner
Goal (User Story): "As a user, I want to search, filter, and sort food items by cost and nutrition so that I can identify the best options for my goals and budget."
Preconditions: At least one `FoodItem` exists with valid derived metrics.
Main Success Scenario:

1. The actor calls `GET /food-items` with query parameters for filtering (e.g., min/max protein per 100 g) and sorting (e.g., price per 100 g protein).

2. The application service builds a database query that uses derived metrics consistently.

3. The API returns a paginated list including the requested sort order and summary fields.

Alternate / Failure Flows:

1F. No matching items → `200 OK` with an empty array.

2F. Unsupported filter or sort parameter → `400 Bad Request` listing allowed fields and formats.

Postconditions: The actor receives a consistently ordered list of food items aligned to their constraints.
Acceptance Criteria:

- Sorting by price-per-protein and other metrics is stable and repeatable.
- Pagination metadata and caching headers are present and correct.

UC06 – Cart Summary and Pack Estimates (Should)

Primary Actor: Shopper preparing purchases
Goal (User Story): "As a shopper, I want a consolidated shopping cart summary with estimated packs and total cost so that I can purchase the right amounts efficiently."
Preconditions: The shopping cart exists and contains one or more items; each referenced `FoodItem` has a known pack size and pack price.
Main Success Scenario:

1. The actor calls `GET /shopping-carts/{cartId}/summary`.
2. The service sums grams per `FoodItem` across all cart items.
3. For each `FoodItem`, the service computes *estimatedPacks* = $\lceil$ `totalGrams/packSizeGrams` $\rceil$ and *estimatedCost* = `estimatedPacks` × `packPrice`.
4. The API returns the consolidated list with totals for cost and macronutrients.

Alternate / Failure Flows:

1F. Unknown `cartId` → `404 Not Found`.

2F. Missing pack size or price for a referenced item → `422 Unprocessable Entity` with guidance on how to fix.

Postconditions: The actor receives a clear summary that can be used for purchasing and budgeting.
Acceptance Criteria:

- Estimated packs are rounded up correctly; cost totals equal the sum of per-item estimates.
- Macro totals across the cart equal the sum of all items within rounding.