BS Portfolio 01 MealPlanner

[Author Name 1] (*[Matriculation Number]*)
[Author Name 2] (*[Matriculation Number]*)

October 27, 2025

1 Project Overview

Elevator Pitch. MealPlanner is a RESTful backend that enables budget-sensitive nutrition planners to curate ingredients, compose dishes, and rank meals by nutritional yield per euro so they can hit protein targets without overspending.

Inflation makes reliable meal planning difficult because prices change faster than spreadsheets. MealPlanner ingests trustworthy ingredient data, recalculates macronutrients and costs whenever dishes change, and exposes ranked endpoints that reveal the best value meals immediately. The MVP delivers CRUD management for `FoodItem` and `Dish` resources, deterministic aggregation in a hexagonal Quarkus service, and paginated, filterable rankings that honour course requirements for REST, JPA, and testing.

Explicit non-goals for the first increment are modelling allergens or micronutrients and introducing user accounts or automated meal-plan generation; these stay out of scope to keep the deliverable focused.

2 Domain Model

The core entities and relationships appear in Figure 1. The domain layer ensures each dish retains at least one ingredient line, and every line records the gram quantity that drives deterministic cost and macronutrient calculations.

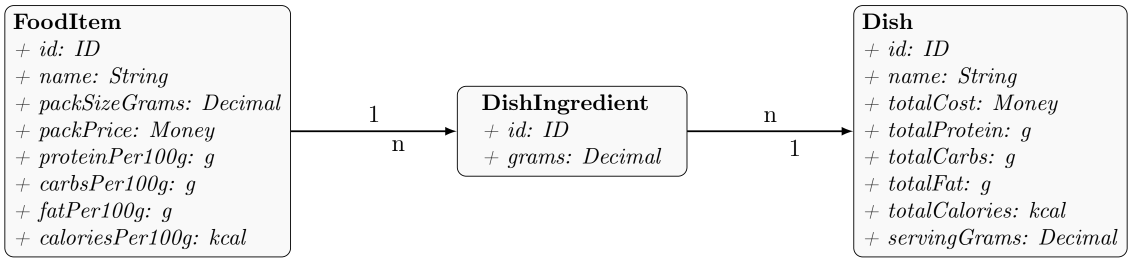

Figure 1: UML class diagram of the Meal Planner domain.

Key entities. **FoodItem** stores its pack size and price alongside nutritional facts per 100 g, allowing the service to derive cost metrics such as price per 100 g. **DishIngredient** records only the grams of a referenced food item; all macro shares are derived on demand. **Dish** aggregates ingredients and persists the latest totals (cost, protein, carbohydrates, fat, calories) together with *servingGrams* so ranking endpoints can query efficiently without recomputing every aggregation on read.

Domain invariants.

- Pack price and size must be positive; macro values per 100 g remain within plausible physiological limits.
- DishIngredient entries require strictly positive grams; operations that leave a dish empty are rejected.
- Aggregated dish totals (cost and macronutrients) are recalculated transactionally in the domain layer by multiplying each `FoodItem` profile with its recorded grams, rounded to two decimals, and persisted together with `servingGrams` to support fast sorting and paging.

Example records.

- FoodItem: *"Chicken Breast"*, pack size 1,000 g, pack price 7,50 €, protein 23 g/100 g, carbohydrates 0 g/100 g, fat 1,5 g/100 g, calories 110 kcal/100 g.
- Dish: *"Budget Protein Bowl"* with 150 g Chicken Breast, 100 g Rice, and 50 g Broccoli, yielding a total cost of 2,45 €, 38 g protein, 58 g carbohydrates, 11 g fat, and 510 kcal.

3 Use Cases

The backend covers four core use cases. Each outlines priority, preconditions, main steps, alternate flows, postconditions, and acceptance criteria.

UC01 – Register Food Item (Must)

Primary Actor: Data curator.
Goal (User Story): "As a curator, I want to register a base ingredient with reliable macro values and pricing so dishes can use authoritative data."
Preconditions: Ingredient name is unique; macro and price fields are present and valid.
Main Success Scenario:

1. The actor submits `POST /food-items` with name, macros per 100 g, unit price, and timestamps.
2. The domain validates ranges, then persists the `FoodItem` via the persistence port.
3. The API returns `201 Created` with the identifier and echo data.

Alternate / Failure Flows:

1F. Duplicate name → `409 Conflict` with guidance to adjust the identifier.

2F. Validation error (e.g., negative macros) → `400 Bad Request` with constraint details.

Postconditions: FoodItem stored and available for dish composition.
Acceptance Criteria:

- Numeric inputs persist with two-decimal precision; negatives are rejected.
- Unit tests cover validation boundaries and duplicate handling.

UC02 – Compose Dish (Must)

Primary Actor: Nutrition planner.
Goal (User Story): "As a planner, I want to assemble a dish from precise ingredient weights so MealPlanner calculates totals automatically."
Preconditions: At least one `FoodItem` exists; gram weights are positive.
Main Success Scenario:

1. The actor calls `POST /dishes` with name, servings, and an ingredient list (`foodItemId`, grams).
2. The application service loads referenced ingredients and recomputes totals and ranking metrics.
3. The transaction persists the `Dish` and `DishIngredient` entities atomically.
4. The API returns `201 Created` with calculated totals and optimistic-lock metadata.

Alternate / Failure Flows:

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

2F. Ingredient grams ≤ 0 → `400 Bad Request` with validation hints.

Postconditions: Dish totals reflect submitted ingredients.
Acceptance Criteria:

- Calculated totals equal the sum of ingredient contributions within rounding tolerance.
- Integration tests confirm persistence and retrieval reproduce totals verbatim.

UC03 – Adjust Dish Composition (Should)

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

1. The actor modifies the composition using sub-resources:
   - 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 reapplies validation and recomputes totals, ratios, and derived rankings.
3. The persistence layer applies changes in a transaction and updates audit timestamps.
4. The API returns the refreshed `Dish` representation with version metadata.

Alternate / Failure Flows:

1F. Dish locked by another transaction → `409 Conflict` with retry advice.

2F. Removing all ingredients → `400 Bad Request`, message "Dish must contain at least one ingredient."

Postconditions: Dish reflects new macro and cost totals.
Acceptance Criteria:

- Integration tests ensure per-serving calculations refresh after modifications.
- Optimistic locking prevents concurrent overwrites without warning.

UC04 – Retrieve Ranked Dishes (Must)

Primary Actor: Shopper preparing meal plans.
Goal (User Story): "As a shopper, I want to list dishes sorted by specific value metrics so I can select the most cost-effective meals quickly."
Preconditions: At least one dish with aggregated metrics exists; pagination defaults are configured.
Main Success Scenario:

1. The actor calls `GET /dishes` with query parameters for sorting, filtering, and paging, e.g.: `/dishes?sort=pricePerProtein:asc,name:asc&page=0&size=20&minProtein=30`

2. The application service issues a database query with the corresponding `ORDER BY` clause and pagination window.

3. The API returns a paginated JSON list with rank numbers, summary metrics, and hypermedia links (`Link` headers or body) to adjacent pages.

Alternate / Failure Flows:

1F. No dishes found → `200 OK` with empty array.

2F. Unsupported sort field → `400 Bad Request` listing valid options.

Postconditions: User receives ordered dish list for immediate comparison.
Acceptance Criteria:

- Sorting stays stable via deterministic tie-breakers (e.g., `name`, then `id`).
- Integration tests assert ordering, filters, pagination metadata, caching headers, and hypermedia navigation.