BS Portfolio 01 MealPlanner

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

October 31, 2025

1 Project Overview

Elevator Pitch. A RESTful Meal Planner backend helps nutrition-focused users curate ingredients and dishes, then ranks meals by nutritional value per cost so they can stretch their food budgets without sacrificing protein targets.

Why it matters. Inflation and rising food prices make it hard for students and athletes to maintain high-protein diets. Today they juggle spreadsheets that rarely stay in sync with grocery prices. This project consolidates reliable ingredient data, automatically aggregates macro nutrients per dish, and exposes query endpoints that surface the best value choices immediately.

Initial capabilities (MVP scope).

- CRUD endpoints for ingredients recorded per 100 g, storing name, price, protein, carbohydrates, fat, and calories.
- Dish management with ingredients and gram-weights, with automatic macro and cost aggregation.
- Read endpoints that return dishes sorted by price, protein density, calorie density, price per 100 g protein, and price per 1000 calories via dedicated resource paths.
- Persistence through Quarkus with JPA, layered using hexagonal architecture (REST adapters, application services, domain core, JPA adapters).
- Automated testing: unit tests for aggregation rules, integration tests covering REST endpoints and database access.

Non-goals (explicit exclusions).

- Modelling allergens, sugar, salt, or micronutrients in MVP (left for a future iteration).
- Personal accounts, authentication, or company-specific branding.
- Recipe recommendation engines or meal-plan generation—focus stays on ranking and manual selection.

2 Domain Model

The core entities and relationships are illustrated in Figure 1. The domain enforces that dishes contain at least one ingredient and each ingredient line stores a gram quantity to derive totals.

Key entities. **FoodItem** stores the canonical nutritional facts per 100 g and current market price. **Dish** aggregates one or more **DishIngredient** records, which capture how much of each ingredient is used along with denormalised totals (ensuring quick read performance when sorting). Totals are recalculated in the domain layer whenever a dish changes.

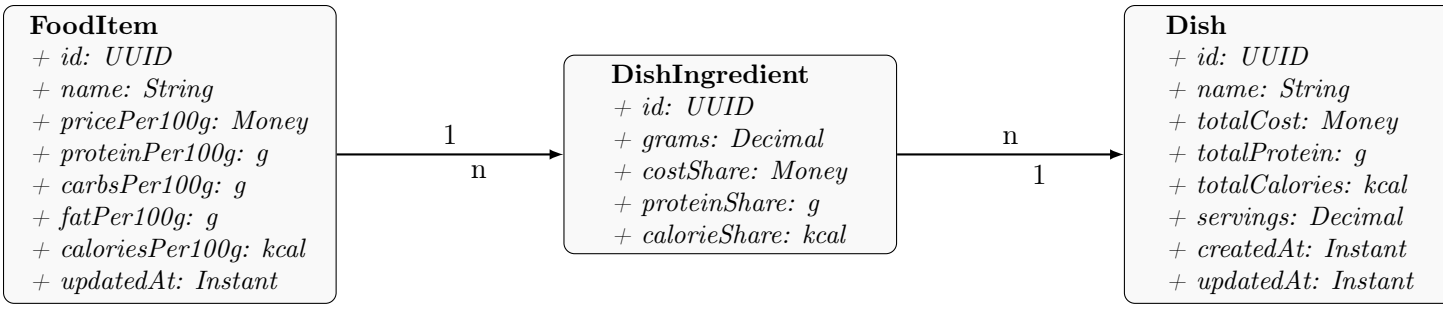

Figure 1: UML class diagram of the Meal Planner domain.

Example records.

- FoodItem: *"Chicken Breast"*, price 1,89 € / 100 g, protein 23 g, carbs 0 g, fat 1,5 g, calories 110 kcal.
- Dish: *"Budget Protein Bowl"* with ingredients (150 g Chicken Breast, 100 g Rice, 50 g Broccoli) resulting in total cost 2,45 €, total protein 38 g, total calories 510 kcal.

3 Use Cases

The project will support the following core use cases. Each includes priority (Must/Should/Could), preconditions, main success path, failure 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 macro values and price so the system can use the data in dishes."
Preconditions: Ingredient name is unique; mandatory macro and price fields provided.
Main Success Scenario:

1. Actor sends `POST /food-items` with name, macros per 100 g, price, and timestamp.
2. System validates ranges (e.g., no negative macros).
3. System persists FoodItem via JPA repository.
4. API returns `201 Created` with generated ID.

Alternate / Failure Flows:

1F. Duplicate name detected → respond `409 Conflict`.

2F. Validation error (e.g., missing protein) → respond `400 Bad Request`.

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

- Numeric values up to two decimals; negative submissions rejected.
- Unit tests cover validation boundaries.

UC02 – Compose Dish (Must)

Primary Actor: Nutritional planner.
Goal (User Story): "As a planner I want to assemble a dish from specific ingredient weights so totals update automatically."
Preconditions: At least one FoodItem exists.
Main Success Scenario:

1. Actor calls `POST /dishes` with name, servings, and ingredient list (`foodItemId`, grams).
2. System loads referenced FoodItems.
3. Domain service calculates aggregated macros, cost, and per-serving metrics.
4. Dish plus DishIngredient entities persist inside a transaction.
5. API returns `201 Created` with calculated totals.

Alternate / Failure Flows:

1F. Unknown FoodItem ID → `404 Not Found`.

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

Postconditions: Dish totals reflect submitted ingredients.
Acceptance Criteria:

- Totals equal sum of ingredient contributions within rounding tolerance.
- Integration test verifies persistence and retrieval reproduces totals.

UC03 – Adjust Dish Composition (Should)

Primary Actor: Nutritional planner.
Goal (User Story): "As a planner I want to update ingredient weights so cost-per-protein stays accurate."
Preconditions: Dish exists with at least one ingredient.
Main Success Scenario:

1. Actor issues `PUT /dishes/{dishId}` with modified ingredient list.
2. System reapplies domain rules to recompute totals and price metrics.
3. Transaction persists new DishIngredient rows, removing obsolete ones.
4. API returns updated DTO with timestamps.

Alternate / Failure Flows:

1F. Dish locked by another transaction → respond `409 Conflict`.

2F. Removal of all ingredients → reject with `400`, message "Dish must contain at least one ingredient."

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

- Integration tests ensure per-serving calculations update after change.
- Optimistic locking guards against concurrent overwrites.

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 so I can pick the most cost-effective meals."
Preconditions: At least one dish exists with aggregated metrics.
Main Success Scenario:

1. Actor calls dedicated endpoints such as:
   - `GET /dishes/sorted-by-price`
   - `GET /dishes/sorted-by-protein`
   - `GET /dishes/sorted-by-calories`
   - `GET /dishes/sorted-by-price-per-protein`
   - `GET /dishes/sorted-by-price-per-calorie`
2. Application service queries database with appropriate `ORDER BY`.
3. Results return as paginated JSON list including ranks and summary metrics.

Alternate / Failure Flows:

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

2F. Unsupported sort path requested → `404 Not Found`.

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

- Sorting stable for dishes with equal metric values.
- Integration tests assert ordering correctness for each endpoint.

UC05 – Inspect Ingredient Impact (Could)

Primary Actor: Planner evaluating substitutions.
Goal (User Story): "As a planner I want to see how a FoodItem contributes to total dish cost and macros so I can replace expensive ingredients."
Preconditions: Dish with recorded DishIngredients.
Main Success Scenario:

1. Actor calls `GET /dishes/{dishId}/ingredients/{ingredientId}`.
2. System returns macro and cost share percentages.

Alternate / Failure Flows:

1F. Ingredient not present in dish → `404 Not Found`.

Postconditions: Planner sees how each ingredient contributes.
Acceptance Criteria:

- Percentages sum to 100% across all ingredients (within rounding tolerance).
- Endpoint disabled in MVP if time constraints demand—marked Could.

4 Testing and Quality Approach

- **Unit tests** target domain services: macro aggregation, price-per-metric calculations, validation rules.
- **Integration tests** spin up Quarkus with an in-memory database to cover REST endpoints, JPA repositories, and JSON contracts.
- **Contract checks** ensure DTOs remain backward compatible when new fields (e.g., micronutrients) are added later.
- **Continuous Integration** enforces formatting, static analysis, and test execution to maintain a releasable trunk.