

Lab 1. RESTful API Specification & Implementation

I chose topic at number 7: *Хмарна інформаційна система для обліку спожитих калорій.*

GitHub link: <https://github.com/SE240-API-Design-25/lab1-rest-MarharytaFilipovych>

Goal*

Develop a comprehensive RESTful API specification and implementation for a cloud-based calorie tracking system that enables users to monitor their daily caloric intake, manage food databases, and track nutritional goals through standardized HTTP endpoints.

Project Tasks*

- Perform system decomposition and identify core modules
- Perform event storming
- Create a Ubiquitous Language for domain entities
- Design a RESTful API specification with proper endpoints and data models
- Implement API with validation, pagination, and error handling
- Develop a presentation layer with CRUD operations
- Add filtering and search capabilities for food items and meal tracking
- Create your own API documentation with YAML

Results

Entities

Product: represents any edible item that you may find and buy in any super(mini)markets, from raw ingredients to ready-to-eat products. Note: complex plates from the culinary department don't count → only those things that were brought from a factory warehouse, village, market, or somewhere they are usually brought from (I do not have a clue).

- `id` UUID → this is a primary key
- `name` string
- `barcode` string value, if this product comes from a factory, I suppose it has to have this thing.
- `proteins` decimal, per 100
- `fats` decimal, per 100
- `carbohydrates` decimal, per 100
- `water` decimal, 0-100%
- `salt` decimal, per 100
- `sugar` decimal, per 100
- `fiber` decimal, per 100
- `alcohol` decimal, per 100
- `description` something that needs to be told about this particular product
- `measurement_unit` I believe this can be represented as an enum value → GRAMS, MILLILITERS, PIECES
- `calories` decimal, per 100 grams
- `created_at` timestamp. This field should not be shown to our end customer, but I think it needs to be included for our developer purposes.
- `brand_id` reference to a product's brand. This field is optional because not all products have a brand. For instance, a plain dirty carrot does not have one.
- `updated_at` timestamp. This field should not be shown to our end customer, but I think it needs to be included for our developer purposes.

Brand: represents a product's brand:

- `id` UUID → this is a primary key
- `name` string value
- `description`

Category: represents a product classification system (such things as dairy, fast food, sweets, chocolate, etc). It is allowed for a product to be included in multiple categories:

- `id` UUID → this is a primary key

- `name` string value
- `description` explanation for those who do not know what dairy, sweets, and vegetables are :
(((
- `parent_category_id` foreign key, for subcategories. (*cloud ai generated*)
- `created_at` timestamp, explained the purpose above

User: my app users. I believe that almost every app needs a user, and mine is not an exception. However, my users have some additional info defined with them since I have a calorie app here.

- `id` UUID → this is a primary key
- `email` string value, I think it has to be unique. This is because in my system this field is one of those things that define the personality.
- `first_name` string value
- `last_name` string value
- `telephone` string value. This is optional because this is not a primary user identifier. By the way, this does not have to be unique.
- `birth_date` date
- `gender` enum → MALE, FEMALE. I will not have UNDEFINED, sorry.
- `weight` decimal in kg
- `height` decimal in cm
- `activity_level` enum → SEDENTARY, LOW, MODERATE, HIGH, VERY_HIGH (*cloud ai generated*)
- `goal` enum → LOSE, MAINTAIN, GAIN)
- `target_weight` decimal, in kg. This is the optional field → maybe my user does not want to loose or gain weight, just support.
- `registered_at` timestamp
- `updated_at` timestamp. This field should not be shown to our end customer, but I think it needs to be included for our developer purposes.

You may wonder, how can we understand how many calories a user can consume in a day? This is the tempting question, I know. The thing is that this field (`daily_calorie_target`) can be calculated from the existing data: goal, gender, weight, age, activity, and height. Therefore, it

should not be stored in the table according to the third normal form. Here is the **formula*** my app uses:

Daily Calorie Target Formula

Step 1: BMR (Basal Metabolic Rate)

- **Men:** $\text{BMR} = (10 \times \text{weight}) + (6.25 \times \text{height}) - (5 \times \text{age}) + 5$
- **Women:** $\text{BMR} = (10 \times \text{weight}) + (6.25 \times \text{height}) - (5 \times \text{age}) - 161$

Step 2: Activity Adjustment

...

$\text{TDEE} = \text{BMR} \times \text{Activity_Factor}$

...

- Sedentary: 1.2
- Low: 1.375
- Moderate: 1.55
- High: 1.725
- Very High: 1.9

Step 3: Goal Adjustment

- **Lose Weight:** $\text{TDEE} - 500$
- **Maintain:** TDEE
- **Gain Weight:** $\text{TDEE} + 500$

Complete Formula:

$\text{daily_calories} = (\text{BMR} \times \text{activity_factor}) \pm \text{goal_adjustment}$

Safety limits: Minimum 1200 (women) / 1500 (men) calories per day.

Plate: represents a combination of products representing a meal or recipe. So, in order to create a plate, one needs to define its name and add a collection of different products with their appropriate measurement to this plate.

- **id** UUID → this is a primary key
- **name** string value
- **description** optional description of what this plate represents. Perhaps a user is lazy and does not wish to provide any description, and this is totally okay
- **user_id** reference to the user who created this plate

- `is_public` boolean. This is false by default, but if the user has a premium account, they can make their plate public and available for usage by others.
- `created_at` timestamp

`total_calories, total_proteins, total_fats, total_carbohydrates, total_quantity` -> these info is calculated when needed.

Record this is about which plate or product a user consumed and recorded, like 50 grams of a self-made chocolate cake or 70 grams of a cucumber on the 30th of May, for lunch.

- `id` UUID → this is a primary key
- `user_id` reference to the hungry user
- `meal_type` enum → BREAKFAST, LUNCH, DINNER, FIRST_SNACK, SECOND_SNACK, THIRD_SNACK
- `consumed_at` timestamp
- `updated_at` timestamp. This field should not be shown to our end customer, but I think it needs to be included for our developer purposes.

The `calories_consumed, total_proteins, total_fats, total_carbohydrates,` and `total_quantity` fields are calculated based on the product record's values per 100. These fields are shown in response only (not stored).

Weight_History* tracks user weight changes over time.

- `id` UUID → this is a primary key
- `user_id` reference to the user
- `weight` decimal in kg
- `recorded_at` timestamp

Junction entities

Product_Category represents a many-to-many relationship between products and categories.

- `product_id` reference to a product
- `category_id` reference to a category
- `quantity` decimal in grams

The primary key is a composite of these 2 IDs.

Plate_Product represents a many-to-many relationship defining which products are in each plate.

Attributes:

- **plate_id** reference to a plate
- **product_id** reference to a product
- **quantity** decimal in grams

The primary key is a composite of these 2 IDs.

Product_record: this is about which product a user consumed and recorded, like 50 grams of a green apple on the 30th of May, lunch.

- **record_id** reference to a record
- **product_id** reference to the consumed (or to-be consumed) product
- **created_at** timestamp
- **quantity** how much of this product was consumed

The primary key is a composite of these 2 IDs.

Plate_Record this is about which plate a user consumed and recorded, like 50 grams of a self-made chocolate cake on the 30th of May, lunch.

Attributes:

- **record_id** reference to a record
- **plate_id** reference to a plate
- **created_at** timestamp
- **quantity** how much of this plate was consumed

The primary key is a composite of these 2 IDs.

Key Relationships*

- **A user can create many plates:** users build their own recipe collections over time
- **A user can log many consumption records:** each time someone eats, they create a new record (this is how I live)
- **A user can have multiple weight entries:** weight is tracked over time to show progress
- **A user belongs to one account:** each person has their own profile with personal goals
- **A brand can have many products:** Coca-Cola makes Coke, Sprite, Fanta, etc.

- **A product may or may not have a brand:** branded items vs. generic produce like "potatoes"
- **A product can belong to multiple categories:** a Greek yogurt is both "dairy" and "protein-rich."
- **A category can have subcategories:** "Dairy" contains "Milk", "Cheese", "Yogurt"
- **A record can contain multiple products:** breakfast might include bread, butter, and jam
- **A record can contain multiple plates:** lunch could be salad + soup (two different recipes)
- **A product appears in many consumption records:** people eat the same apple brand repeatedly
- **A plate appears in many consumption records:** users repeat their favorite recipes
- **A plate contains multiple products as ingredients:** Caesar salad needs lettuce, croutons, dressing, and parmesan
- **A product can be used in multiple dishes:** eggs go in pancakes, omelets, cakes, etc.
- **A plate belongs to one creator:** each recipe has an owner who designed it
- **A plate can be shared publicly:** premium users can make their recipes available to others

Explicit explanation of relationships:*

- **User (1) → Plate (N)** - One user creates many plates
- **User (1) → Record (N)** - One user logs many consumption events
- **User (1) → Weight_History (N)** - One user has many weight measurements over time
- **Brand (1) → Product (N)** - One brand manufactures many products
- **Category (1) → Category (N)** - One category contains many subcategories
- **Record (1) → Product_Record (N)** - One meal record contains many individual products
- **Record (1) → Plate_Record (N)** - One meal record contains many plates/recipes
- **Product (1) → Product_Record (N)** - One product appears in many consumption events
- **Plate (1) → Plate_Record (N)** - One recipe is consumed many times by users
- **Product (M) ↔ Category (N)** via Product_Category junction. Many products belong to many categories

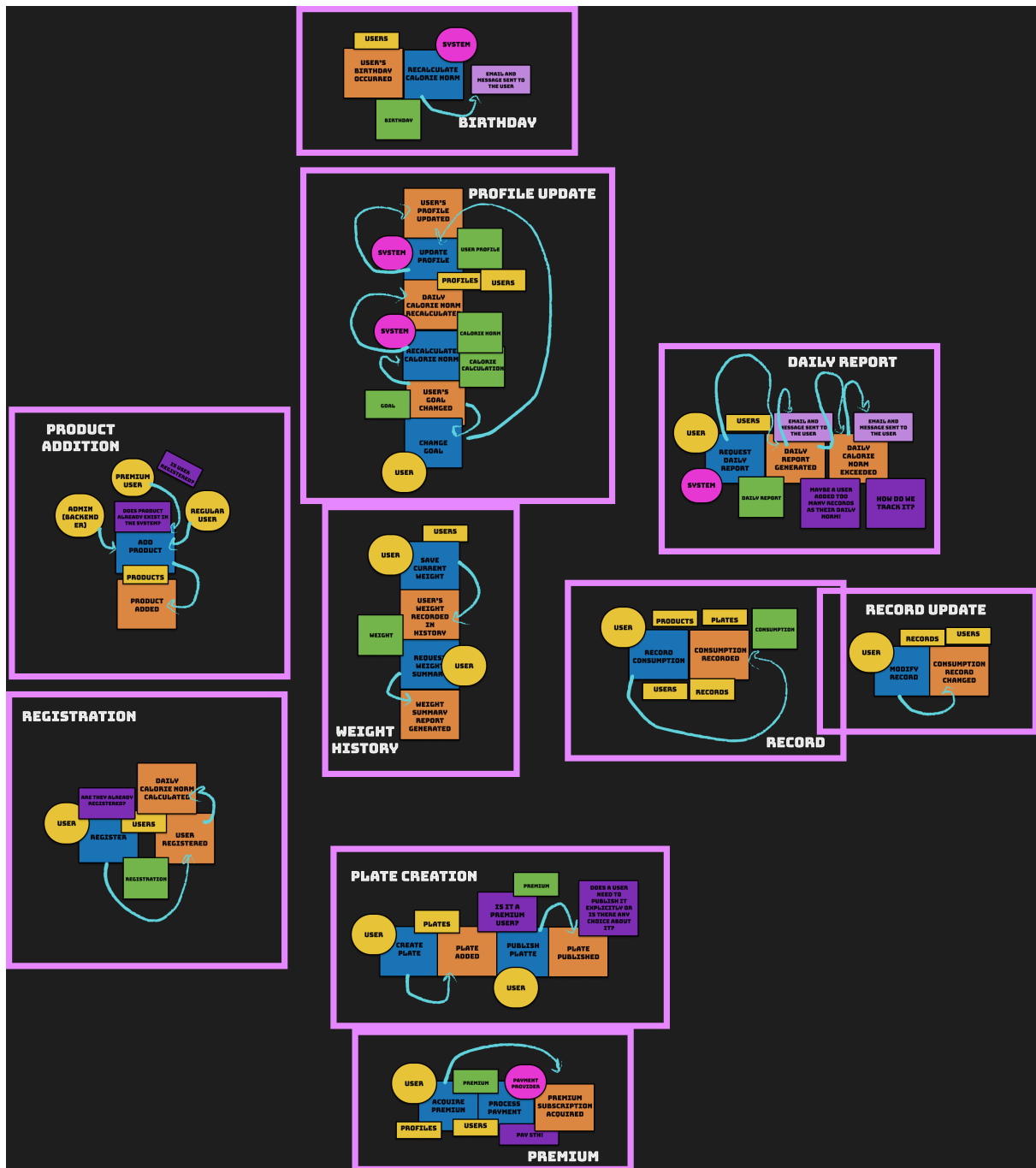
- **Product (M) ↔ Plate (N)** via Plate_Product junction. Many products are ingredients in many recipes. Many recipes contain many products
- **Brand (0:1) → Product (N)** - Products may or may not have a brand (carrots don't, Coca-Cola does)

Ubiquitous language:

- **per 100** - means that a particular attribute represents a quantity per 100 of a particular measurement (like 100 g or 100 ml)
- **product** - anything that you may get in the market, but not sth that was prepared
- **plate** - a combination of such products
- **daily report** - a report about the quantity of all nutrients, products (or plates), and calories eaten during a particular day
- **admin (backender)** - people who manage this system and add products. Products can be added either by a user or by an admin. If a product is added by a user, it becomes visible and usable to all other users without a premium.
- **update profile** - in the current context, it means changing body details (height, weight, birthday, etc.), which impacts daily calorie norm
- **weight summary** and **weight history** represent the same thing
- **premium** - premium account

Event storming

<https://www.figma.com/design/kYlvz2Dwpqq8ePjTY6ciGJ/Event-storming?node-id=0-1>



Context Mapping and Module Decomposition*

User Management Context handles user lifecycle operations including registration, profile management, and authentication. This context owns the User aggregate and manages personal information, physical characteristics, and fitness goals. Key events include User Registered, Profile Updated, and Daily Calorie Norm Calculated, with the system automatically recalculating calorie targets when user parameters change.

Product Catalog Context manages the food database including products, brands, and categories. This context supports both admin-managed and user-contributed products, with

validation ensuring data quality. Core events include Product Added, Product Updated, and Product Archived, enabling a comprehensive food database that serves the entire application.

Consumption Tracking Context represents the core domain functionality, handling meal logging and nutritional calculations. This context orchestrates consumption records, integrating products and plates into daily meal tracking. Primary events include Consumption Recorded, Record Updated, and Daily Report Generated, with complex business logic for calculating nutritional totals and validating consumption data.

Recipe Management Context manages user-created plates and recipes, supporting both private and public sharing for premium users. Events include Plate Created, Plate Published, and Recipe Shared, enabling users to build reusable meal combinations and share successful recipes with the community.

Analytics Context provides reporting and progress tracking functionality, generating daily summaries and weight history analysis. Key events include Daily Report Generated, Weight History Updated, and Progress Calculated, delivering insights into user consumption patterns and goal achievement.

Premium Context handles subscription management and premium feature access, controlling which users can publish public plates and access advanced analytics. Events include Premium Activated, Subscription Renewed, and Premium Features Unlocked ensure proper feature gating and billing integration.

Description of API specification

I defined my specification in the **api_documentation.yaml** file. Here it is:

[api_documentation.yaml](#)

Short overview*

This is a cloud-based RESTful API designed for comprehensive caloric intake and nutritional tracking. The system revolves around three core entities: Products containing detailed nutritional information including calories and nine different nutritional components, Users with complete profiles encompassing physical statistics and dietary goals, and Records that log consumption events linking users to specific products with quantities and meal categorization.

The API provides full CRUD operations for both products and users, while consumption tracking allows users to log meals, update existing records, and view their consumption

history. A notable feature is the automatic daily calorie target calculation based on individual user profiles including weight, height, activity level, and goals. The system supports comprehensive search and filtering capabilities, enabling product searches by name and date-based filtering for consumption records.

Following RESTful principles, the API uses standard HTTP methods and status codes with a hierarchical design where records are nested under users. All list endpoints implement consistent pagination using limit and offset parameters. The system employs UUID identifiers for all entities and uses JSON for all request and response formats. Input validation is comprehensive with detailed error responses, and the API supports flexible meal categorization across six different meal types including breakfast, lunch, dinner, and three snack categories.

Products Management

- `GET /products` - List products with filtering by name and pagination
- `POST /products` - Create new product
- `GET /products/{id}` - Get specific product details
- `PUT /products/{id}` - Update existing product
- `DELETE /products/{id}` - Remove product

User Management

- `GET /users` - List all registered users (paginated)
- `POST /users` - Register new user
- `GET /users/{id}` - Get user profile
- `PUT /users/{id}` - Update user profile
- `DELETE /users/{id}` - Remove user
- `GET /users/email/{email}` - Find user by email

Calorie Calculation

- `GET /users/{id}/daily-target` - Calculate daily calorie target for user

Consumption Records

- `GET /users/{userId}/records` - Get user's consumption history (filterable by date)
- `POST /users/{userId}/records` - Log new meal consumption

- `PUT /users/{userId}/records/{id}` - Update consumption record
- `DELETE /users/{userId}/records/{id}` - Delete consumption record

Key Features

- **Hierarchical API:** Records are nested under users (`/users/{userId}/records`)
- **Comprehensive Product Data:** Detailed nutritional information (proteins, fats, carbs, water, salt, sugar, fiber, alcohol)
- **Flexible Meal Types:** 6 meal categories (breakfast, lunch, dinner, 3 snack types)
- **User Profiles:** Complete user data including physical stats, activity level, and weight goals
- **Pagination:** Consistent limit/offset pagination across list endpoints
- **Filtering:** Product search by name, record filtering by date

App code description*

For the code implementation, four core entities were selected: **User**, **Product**, **Record**, and **ProductRecord** (serving as a junction entity linking products to consumption records with quantities). These entities form the essential foundation for user management, food database, and consumption tracking functionality, representing the minimum viable product for a calorie-tracking application. Other entities were omitted for simplicity.

Presentation layer

The presentation layer serves as the interface boundary of this Spring Boot calorie tracking application, handling HTTP requests, data validation, response formatting, and implementing comprehensive RESTful API controllers.

The layer defines five core enums that represent the application's domain concepts: **ActivityLevel** (from sedentary to very high), **Gender** (male/female), **Goal** (lose/maintain/gain weight), **MealType** (breakfast through dinner plus three snack options), and **MeasurementUnit** (grams, milliliters, pieces). These enums provide type safety and constrain valid values throughout the application.

A sophisticated validation system is built around three custom annotations. The `@CorrectEnum` annotation validates enum values using the `EnumValidator` class, ensuring only valid enum instances are accepted. The `@CorrectName` annotation employs `NameValidator` to enforce name formatting rules, requiring 2-150 character names containing only letters, spaces, hyphens, apostrophes, and periods, with optional fields supported. The

@Nutrient annotation uses NutrientValidator to validate nutritional values, ensuring they fall between 0-100 with customizable nutrient names and measurement units in error messages.

The LogAspect class provides comprehensive logging through aspect-oriented programming. It intercepts all controller methods to log HTTP requests, parameters, execution times, and response status codes. Additionally, it captures and logs exceptions from both service and repository layers, providing complete visibility into application behavior and error tracking.

The presentation layer includes utility classes for consistent API responses. The ErrorResponse record provides standardized error messaging, while the Meta class encapsulates pagination metadata extracted from Spring Data Page objects, including page numbers, total counts, and navigation flags. The Pagination class defines request parameters for paginated endpoints with validation constraints limiting page sizes between 1-100 items.

The controllers implement a comprehensive RESTful API structure handling all aspects of the calorie tracking application through three primary controller classes.

The UserController serves as the primary interface for user lifecycle management, implementing a complete set of RESTful endpoints for user operations. The controller is annotated with @RestController and @RequestMapping("/api/users") to establish the base path for all user-related operations. A GET endpoint at the root path handles user listing with pagination support, accepting Pagination objects as @ModelAttribute parameters to enable query parameters like ?page=1&limit=20. The method delegates to UserService.getAllUsers() and returns a ResponseEntity containing a Page<UserDto> wrapped in a standardized response format with embedded Meta pagination information.

User registration is handled through a POST endpoint at the root path, accepting a @Valid @RequestBody UserDto parameter to trigger comprehensive validation including email format validation, name validation through @CorrectName, enum validation for gender, activity level, and goals, plus range validation for physical measurements. The controller catches validation exceptions and returns structured error responses using the ErrorResponse record. Upon successful validation, it calls UserService.createUser() and returns a 201 Created status with the newly created user's UUID.

Individual user operations include GET("/{id}") for user profile retrieval using @PathVariable UUID id, with the controller handling EntityNotFoundException from the service layer and converting it to a 404 Not Found response. User profile updates use a PUT endpoint at("/{id}") accepting both path variable and request body, implementing complete resource replacement semantics through UserService.updateUser(). A DELETE endpoint at("/{id}") handles user removal by calling UserService.deleteUser().

A specialized GET endpoint at("/{id}/daily-target" exposes the calculated daily calorie target through UserService.getDailyTarget(), demonstrating how controllers can expose computed

business logic through clean REST interfaces. An additional GET endpoint at `"/email/{email}"` supports user lookup by email address using `UserService.getUserByEmail()`, providing an alternative access pattern for user identification.

The `ProductController` manages the extensive food product database through a sophisticated set of endpoints designed for both administrative product management and user product discovery. The controller is mapped to `"/api/products"` and implements comprehensive CRUD operations with advanced querying capabilities.

The primary GET endpoint supports optional query parameters including name-based search, with the controller method signature accepting `@RequestParam(required = false) String name` alongside pagination parameters extracted from a `Pagination` object. This demonstrates Spring's flexible parameter binding where missing query parameters result in null values passed to the service layer. The search functionality integrates with `ProductService.getAll()` to enable efficient product discovery through database-level filtering rather than application-level filtering.

Product creation through POST implements sophisticated validation combining standard Bean Validation annotations with custom validators. The `@Valid @RequestBody ProductDto` triggers validation of nutritional values through the `@Nutrient` annotation, barcode format validation with `@Size` constraints, and measurement unit enum validation through `@CorrectEnum`. The controller handles the business logic exception for duplicate barcodes by catching `IllegalArgumentException` and returning a 409 Conflict status with descriptive error messages derived from the service layer's validation logic.

Individual product operations include GET `"/{id}"` for product retrieval through `ProductService.getById()` with automatic `EntityNotFoundException` to 404 Not Found conversion, PUT `"/{id}"` for complete product updates with validation through `ProductService.updateProduct()`, and DELETE `"/{id}"` implementing intelligent deletion logic. The delete operation demonstrates complex business rules where the controller delegates to `ProductService.delete()`, which implements either hard deletion for unused products or soft deletion (archiving) for products referenced in consumption records through the `ProductRecordRepository.existsByProduct_Id()` check.

The `RecordController` implements the most complex controller logic, managing meal logging and consumption tracking with sophisticated data aggregation and filtering capabilities. Mapped to `"/users/{userId}/records"`, this controller handles the core functionality of the calorie tracking application with user-scoped operations through a dedicated `RecordService`.

The primary record creation endpoint is a POST at the root path accepting `@PathVariable` UUID `userId` and `@Valid @RequestBody RecordRequestDto`. The `RecordRequestDto` contains nested validation through the `@Valid` annotation on the

Set<ProductRecordInRequestDto> products field, which itself validates product IDs and quantities with @DecimalMin constraints. This demonstrates Spring's deep validation capabilities where validation cascades through object graphs. The controller calls RecordService.createRecord() with the user ID and validated request data, returning the newly created record's UUID with a 201 Created status.

Record retrieval is implemented through a sophisticated GET endpoint supporting multiple query parameters for filtering. The method signature includes @PathVariable UUID userId, @RequestParam(required = false) LocalDate date for date-based filtering (with @PastOrPresent validation), and pagination parameters through a Pagination object. This endpoint showcases Spring's automatic type conversion where string date parameters are converted to LocalDate objects through registered converters. The controller calls RecordService.getRecords() with all parameters, returning a paginated list of RecordResponseDto objects with calculated nutritional totals.

Individual record operations include GET("/{userId}/records/{id}") for detailed record viewing through RecordService.getConsumption(), which triggers the complex calculateRecordTotals() business logic to compute total calories, proteins, fats, carbohydrates, and total quantity consumed. The response includes the complete record with embedded product information and all calculated nutritional summaries. PUT("/{userId}/records/{id}") handles meal modifications through RecordService.updateRecord(), accepting a RecordRequestDto and implementing complete record replacement including product list updates. DELETE("/{userId}/records/{id}") removes consumption records through RecordService.deleteRecord() with proper user ownership validation.

All controllers implement consistent error handling through a global @ControllerAdvice class that catches common exceptions and converts them to appropriate HTTP responses. EntityNotFoundException maps to 404 Not Found responses, IllegalArgumentException maps to 400 Bad Request for business rule violations (like duplicate emails or barcodes), and MethodArgumentNotValidException produces 422 Unprocessable Entity responses with detailed field-level error information extracted from BindingResult.

The controllers leverage Spring's automatic content negotiation to support JSON request/response handling through Jackson serialization. The DTOs include @JsonProperty annotations for field naming control (like "first_name" vs "firstName") and @JsonInclude annotations for conditional field inclusion. Jackson automatically serializes enum values and handles LocalDateTime formatting for timestamps.

Response formatting is standardized across all endpoints, with successful responses containing data payloads and pagination metadata through the Meta class where applicable, while error responses consistently use the ErrorResponse record. HTTP status codes are precisely implemented following REST conventions: 200 for successful retrieval, 201 for

creation, 204 for successful deletion, 400 for validation errors, 404 for missing resources, and 409 for business rule conflicts.

Data Layer

The data layer implements the persistence architecture using JPA entities and Spring Data repositories. Four main entities define the domain model: the User entity stores personal information and fitness goals with enum fields for gender, activity level, and goal type; the Product entity represents food items with comprehensive nutritional information including macronutrients, micronutrients, calories, and measurement units; the Record entity captures meal consumption events linked to users with meal type and timestamp; and the ProductRecord entity serves as a join entity connecting products to records with quantity information. The entities use UUID primary keys, JPA auditing for timestamps, and bidirectional relationships with proper cascade configurations.

Repository interfaces extend JpaRepository providing standard CRUD operations plus custom query methods. UserRepository handles user management with email-based lookups and existence checks. ProductRepository provides product search capabilities with name filtering and archived status management through methods like findProductByNameContainingIgnoreCaseAndArchivedIsFalse(). RecordRepository implements complex querying with date-based filtering and user-scoped operations using @Query annotations for optimized data retrieval like findIdsByUserIdAndDateTime(). ProductRecordRepository manages the join table operations with methods for checking product usage in consumption records.

Business Layer

The business layer contains the service classes that orchestrate business logic and coordinate between the presentation and data layers, now properly separated into focused service responsibilities.

UserService handles user management including registration, profile updates, and daily calorie target calculations using Harris-Benedict equations. Key methods include createUser() for user registration with email uniqueness validation, getUserById() and getUserByEmail() for user retrieval, updateUser() for profile modifications, deleteUser() for user removal, and getDailyTarget() which implements the complex calorie calculation algorithm considering age, gender, weight, height, activity level, and fitness goals.

RecordService is a dedicated service class that manages all record-related operations, separated from UserService for better separation of concerns. It handles meal logging and consumption tracking with sophisticated data aggregation. Key methods include:

- createRecord() for creating new consumption records with product validation and association

- `getRecords()` for retrieving paginated records with optional date filtering using optimized query patterns
- `getConsumption()` for retrieving individual records with complete nutritional calculations
- `updateRecord()` for modifying existing records with `@Transactional` support ensuring data consistency
- `deleteRecord()` for removing consumption records with proper user ownership validation
- `calculateRecordTotals()` for computing total calories, proteins, fats, carbohydrates, and quantities

The service implements complex business logic including product existence validation through `getExistingProducts()`, `ProductRecord` creation and management through `populateProductRecords()`, and comprehensive nutritional calculations with proper rounding and unit conversions.

`ProductService` manages the food database with search capabilities, validation for duplicate barcodes, and intelligent deletion logic implementing soft deletion for referenced products through methods like `create()`, `getById()`, `updateProduct()`, and `delete()`.

The layer uses `MapStruct` mappers for seamless conversion between entities and DTOs. `ProductMapper` handles entity-DTO conversion with automatic calorie calculation using the `calculateCalories()` method and enum mapping between presentation and data layer enums. `UserMapper` provides bidirectional conversion between `User` entities and `UserDto` objects with proper enum mappings for gender, activity level, and goals. `RecordMapper` converts `Record` entities to `RecordResponseDto` objects, integrating with `ProductMapper` to populate nested product information and enabling the complex nutritional calculations performed by the service layer.

Services implement transactional operations and business rule validation, throwing appropriate exceptions like `EntityNotFoundException` for missing resources and `IllegalArgumentException` for business rule violations that are handled by the presentation layer's error management system.

Conclusion*

This laboratory work successfully implemented a RESTful API for a calorie tracking application using Spring Boot with a well-structured three-layer architecture. The presentation layer provides comprehensive request handling through custom validation annotations, sophisticated error management, and complete CRUD operations via `UserController`, `ProductController`, and `RecordController`. The business layer orchestrates domain logic through focused service classes - `UserService` for user management,

RecordService for consumption tracking, and ProductService for food database management - while MapStruct mappers ensure clean entity-DTO conversions. The data layer demonstrates proper JPA design with optimized repositories and custom query methods.

The application adheres to REST principles with appropriate HTTP status codes, resource-oriented URLs, and standardized JSON responses. Key achievements include advanced validation cascading, aspect-oriented logging, intelligent soft deletion, hierarchical resource relationships, and proper separation of concerns through dedicated service classes. The implementation successfully balances functionality with maintainability through consistent architectural patterns and clean separation of responsibilities.

AI usage:

- all headers highlighted in pink color with * indicate sections generated by Cloud AI. Hence, almost all descriptions (conclusion, goal, task objectives/ code description) were generated.
- For the code implementation, I used AI only to find sth new for myself and for the error (bug) handling (obviously), since I started learning the Spring Framework recently. Whenever AI generated sth for me, I did not copy and paste it without thinking. Unfortunately, it is difficult for me to point out what was purely generated (except for standard error messages), because there was no situation in which I asked him to purely generate a code section for me. If you ask me about a specific part of my code, I will tell you what I was helped with if needed. :))
- used it for bug handling (in api doc as well)
- calories and nutrient-related calculations
- help with understanding on how to load data (which is with lazy fetch type)