Lab 4😘 - APIs for AIs

Link: https://github.com/SE240-API-Design-25/lab4-ai-api-MarharytaFilipovych.git

Goal

The **goal** of this lab is to figure out how to help AIs use our APIs U can use OpenAI playground to experiment.

Task

1. Allow LLM to use your REST client developed in CA-1
    - add guardrails on token expenditure
    - add multiple tools. Invent a prompt that expenditure requires chaining them
    - add quadrail against prompt injection
2. Advertise the capabilities of your restful API from lab1 as an MCP server. Use the inspector to verify your work

Results

What Was Implemented

My calorie tracking application demonstrates two distinct yet complementary approaches to AI integration, each serving different architectural needs and use cases within the nutrition management ecosystem.

Function Calling Implementation (Calorie-Client)

The **calorie-client** application implements a sophisticated **AI client architecture** that acts as an intelligent intermediary between users and your nutrition tracking backend. This implementation features:

- **Custom OpenAI Integration** with Groq API using the gemma2-9b-it model
- **Multi-round Function Calling** supporting complex conversational workflows with up to 5 execution rounds
- **Comprehensive Security Framework** including prompt injection detection, input sanitization, and risk scoring
- **Advanced Rate Limiting** with token usage tracking and request throttling
- **Eight Function Tools** providing complete nutrition tracking capabilities through structured JSON schemas
- **Reactive Architecture** using Spring WebFlux for scalable, non-blocking operations
- **Detailed Prompt Engineering** enforcing structured response formats for transparency

This implementation creates an **intelligent nutrition assistant** that can understand natural language requests and execute complex multi-step operations while maintaining security, performance, and reliability standards.

MCP Implementation (Calories Core)

The **Calories** core application implements a **standardized MCP server** that exposes your entire nutrition tracking platform as discoverable AI tools. This implementation features:

- **Spring AI MCP Server** providing standardized Model Context Protocol compliance
- **Four Domain-Specific Tool Classes** (Products, Users, Records, Brands) with 20+ available operations
- **Rich Metadata Annotations** using `@Tool` and `@ToolParam` for intelligent tool discovery
- **Comprehensive CRUD Operations** covering all aspects of nutrition tracking and user management
- **Pagination and Response Standardization** through `PageResponse` for consistent data handling
- **Service Layer Integration** maintaining business logic consistency across access methods
- **Self-Documenting API** enabling AI clients to understand capabilities without external documentation

This implementation creates a **universal nutrition tracking API** that any MCP-compatible AI system can discover, understand, and utilize for sophisticated nutrition management tasks.

Together, these implementations demonstrate **complementary AI integration strategies**:

- The **MCP server** provides a **standardized, discoverable interface** that external AI systems can consume
- The **function calling client** offers a **customized, conversational experience** with advanced security and user interaction features
- Both leverage your **existing service layer** ensuring consistent business logic and data integrity
- The **dual approach** enables both **internal AI assistance** (client) and **external AI integration** (MCP server)

Code explanation

Function calling implementation in calorie-client app

The calorie-client application implements a sophisticated function-calling system that enables an AI nutrition assistant to interact with my calorie-tracking backend through structured tool execution. The implementation centers around the `CalorieLLMService` as the orchestration layer, which integrates multiple specialized services to create a secure, rate-limited, and reliable AI-powered interface.

The foundation of the function calling system rests on the `CalorieLLMService`, which serves as the main orchestrator for processing user messages and managing the entire function calling lifecycle. The service implements a reactive pattern using Spring WebFlux's Mono wrapper around the synchronous OpenAI operations. The core method `processUserMessage(String userMessage)` returns `Mono<ChatResponse>` and coordinates between the OpenAI client for language model interactions, the `CalorieFunctionToolService` for actual function execution, the `PromptInjectionGuardService` for security validation, and the `TokenUsageTrackerService` for rate limiting and usage monitoring.

The `CalorieLLMService` uses constructor injection with `@RequiredArgsConstructor` to wire dependencies, including the crucial `Supplier<ChatCompletionCreateParams.Builder>` that provides pre-configured OpenAI request builders with all the function tools registered. The service maintains stateless operation by tracking function execution state within individual request contexts rather than maintaining instance variables.

The `OpenAIConfig` establishes the connection to the Groq API using the `gemma2-9b-it` model, configured through comprehensive LLM settings that include token limits, request rates, and temperature controls. The configuration automatically registers eight distinct nutrition-related functions including `SearchProducts`, `CreateUser`, `GetUser`, `CreateFoodRecord`, `GetUserRecords`, `GetSingleRecord`, `UpdateFoodRecord`, and `DeleteFoodRecord`, each designed to handle specific aspects of calorie tracking and user management.

The implementation leverages the OpenAI Java SDK's tool registration system through explicit class-based registration in the `OpenAIConfig`. The `chatCompletionBuilderSupplier()` method creates a builder configuration that registers eight function classes using the `addTool(Class.class)` method:

```
.addTool(SearchProducts.class)
.addTool(CreateUser.class)
.addTool(GetUser.class)
.addTool(CreateFoodRecord.class)
.addTool(GetUserRecords.class)
.addTool(GetSingleRecord.class)
.addTool(UpdateFoodRecord.class)
.addTool(DeleteFoodRecord.class)
```

Each function tool is defined as a POJO with extensive Jackson annotations that provide structured metadata for the AI model. The system uses two primary annotation types: `@JsonClassDescription` at the class level provides overall function purpose documentation, while `@JsonPropertyDescription` at field level describes individual parameters.

Function calling execution occurs through the `callFunction(ChatCompletionMessageToolCall.Function function)` method which uses a switch expression to route function calls.

The implementation supports sophisticated multi-round function calling through a controlled loop mechanism in the `processWithOpenAI(String userMessage)` method. The core logic maintains conversation state by building message chains that include the original user request, assistant responses with tool calls, and tool execution results. The method signature demonstrates the reactive approach.

Each round follows a structured execution pattern that checks for tool calls in the assistant response, executes any requested functions, and prepares for the next iteration.

The `executeToolCalls` method processes multiple tool calls within a single round, handling validation, execution, and error recovery.

This enables complex workflows where the AI can chain multiple operations together, such as creating a user profile and then immediately searching for products or logging food consumption. The round-based execution is governed by a configurable maximum rounds limit (set to 5 in your configuration), preventing infinite loops while allowing sufficient iterations for complex multi-step operations.

The implementation includes sophisticated data transformation logic that maps between the AI function model objects and your backend API DTOs. The `CalorieFunctionToolService` contains several private helper methods that handle this transformation, including enum mapping, date parsing, and object construction.

The implementation incorporates robust security measures through the `PromptInjectionGuardService`, which implements a multi-layered validation system designed to prevent prompt injection attacks and malicious input. The service constructor initializes regex patterns from configuration.

The main validation method `validateInput(String input)` implements cumulative risk scoring based on multiple threat detection mechanisms.

The service maintains an extensive list of suspicious patterns configured in `application.yaml` including common injection keywords like "ignore," "forget," "disregard," system-level commands, and persona manipulation attempts. Each pattern type has associated weights that contribute to the overall risk score.

Input sanitization `sanitizeInput(String input)` removes control characters and potentially dangerous patterns while preserving legitimate user input:

The `TokenUsageTrackerService` implements comprehensive usage controls using atomic counters and scheduled reset mechanisms. The service maintains thread-safe tracking of both request frequency and token consumption.

The `canMakeRequest(int estimatedTokens)` method performs pre-flight validation against configured limits.

Token estimation occurs using a simple heuristic `int estimateTokens(String text)` in the `CalorieLLMService`.

Actual token usage tracking occurs after API responses using the `CompletionUsage` metadata:

The recording method `recordUsage(long tokensUsed)` updates atomic counters with comprehensive logging.

The system prompt engineering enforces a structured response format that ensures users receive comprehensive reports of all executed operations. The mandatory response format includes operation numbering, action descriptions, input parameters, specific results, and additional details for each function call. This approach transforms the AI from a simple question-answering system into a detailed operational assistant that provides transparency about all performed actions.

The formatting requirements include specific Unicode symbols for visual organization, ensuring that multi-step operations are clearly presented to users. Summary sections provide high-level outcomes and suggested next steps, creating a conversational experience that guides users through complex nutrition tracking workflows. Error reporting maintains the same structured format, providing clear feedback when operations fail and suggesting alternative approaches.

MCP (Model Context Protocol) Implementation in Calorie App

The calorie application implements a comprehensive Model Context Protocol (MCP) server that exposes the entire nutrition tracking functionality as a standardized AI tool interface. The implementation leverages Spring AI's MCP server capabilities to create a fully-featured nutrition management API that can be consumed by any MCP-compatible AI client, providing seamless integration with large language models and AI assistants.

The MCP implementation centers around Spring AI's `spring-ai-starter-mcp-server-webmvc` dependency, which provides the foundational infrastructure for exposing my application as an MCP server. The core configuration in `McpConfig` establishes the tool callback provider that registers all available nutrition-related operations as MCP tools.

The `MethodToolCallbackProvider` automatically scans the provided tool objects for methods annotated with `@Tool`, creating a comprehensive function registry that MCP clients can discover and invoke. This approach enables automatic tool registration without manual configuration, ensuring that any new methods added to the tool classes are immediately available to MCP clients.

The MCP server configuration in `application.properties` defines the server identity and capabilities.

This configuration establishes the server name as "calorie-tracker-api" and provides descriptive instructions that help AI clients understand the server's purpose. The notification settings are disabled, indicating a stateless API design focused on direct tool invocation rather than real-time change notifications.

My MCP implementation organizes functionality into four distinct tool classes, each responsible for a specific domain within the nutrition tracking ecosystem. This separation follows domain-driven design principles and ensures clear responsibility boundaries while maintaining cohesive functionality groupings.

The **`ProductMcpTools`** class handles all product-related operations, providing comprehensive CRUD functionality for food products including nutritional information management, brand associations, and measurement unit specifications. The **`UserMcpTools`** class manages user account operations, profile management, and calorie target calculations. The **`RecordMcpTools`** class focuses on consumption tracking, meal logging, and historical data retrieval. The **`BrandMcpTools`** class provides brand management capabilities for organizing products by manufacturer or supplier.

Each tool class follows consistent patterns for method signatures, parameter documentation, and error handling, creating a uniform experience for AI clients regardless of which domain functionality they're accessing.

My implementation extensively uses Spring AI's annotation system to provide rich metadata that enables intelligent tool discovery and parameter understanding by AI clients. The `@Tool` annotation defines each exposed method as an MCP tool with detailed descriptions that guide AI decision-making.

The `@ToolParam` annotation provides parameter-level documentation that includes data type specifications, requirement indicators, and behavioral descriptions. This metadata enables AI clients to understand parameter purposes, validate inputs, and generate appropriate function

calls based on user intent.

Complex parameter objects like `ProductRequestDto` and `UserDto` include embedded `@ToolParam` annotations at the field level, providing granular documentation for nested object structures.

This approach creates self-documenting data structures that AI clients can introspect to understand complex parameter requirements and construct valid requests.

The `ProductMcpTools` class exposes comprehensive product management functionality through five core operations that cover the complete lifecycle of food product data. The tool provides intelligent product discovery through the `getProducts` method, which supports both name-based filtering and pagination for handling large product catalogs.

Product creation functionality through `createProduct` enables comprehensive nutritional data management including macronutrient profiles, micronutrient content, measurement units, and brand associations. The method accepts a `ProductRequestDto` that encapsulates all necessary product information, including calories, proteins, fats, carbohydrates, and various micronutrients like salt, sugar, fiber, and alcohol content.

Individual product retrieval via `getProductById` provides detailed product information access using UUID-based identification, ensuring precise product referencing for consumption logging and nutritional analysis. Product updates through `updateProduct` support partial modifications, allowing AI clients to modify specific product attributes without affecting unchanged properties.

The product deletion capability through `deleteProduct` provides data lifecycle management with proper cleanup and referential integrity maintenance through the underlying service layer.

The `UserMcpTools` class implements comprehensive user account management that enables complete user lifecycle operations within the nutrition tracking system. The user creation functionality supports detailed profile establishment, including demographic information, physical characteristics, activity levels, and weight management goals:

User discovery capabilities include both ID-based and email-based lookup methods, providing flexible user identification options for different use cases. The `getUserById` method enables direct user access when the system has UUID references, while `getUserByEmail` supports natural language interactions where users provide email addresses as identification.

The user profile update functionality allows comprehensive profile modifications including weight changes, goal adjustments, and activity level updates, essential for maintaining accurate calorie targets and nutritional recommendations over time.

A specialized calorie target calculation tool
```
int calculateDailyCalorieTarget(
    @ToolParam(description = "User unique identifier (UUID)") UUID userId)
```
provides intelligent daily calorie recommendations based on user profiles:

This method implements sophisticated algorithms that consider age, gender, weight, height, activity level, and weight goals to provide personalized daily calorie targets, enabling AI clients to provide tailored nutritional guidance.

The `RecordMcpTools` class provides comprehensive consumption tracking capabilities that enable detailed meal logging, historical analysis, and nutritional monitoring. The record retrieval functionality supports both general consumption history and date-specific filtering.

The method includes intelligent date parsing that supports ISO date formats while providing null-safe operation for general history retrieval. Pagination support ensures efficient handling of extensive consumption histories without performance degradation.

Consumption record creation enables detailed meal logging with product-quantity relationships and meal type categorization.

Individual record retrieval, updates, and deletion provide complete record lifecycle management, enabling AI clients to help users correct logging errors, update portion sizes, or remove incorrectly logged meals.

The `BrandMcpTools` class provides brand management functionality that supports product organization and manufacturer tracking within the nutrition database. Brand discovery includes both general listing with pagination and specific brand lookup by name or ID.

Brand creation, update, and deletion capabilities enable comprehensive brand lifecycle management, supporting scenarios where users encounter new food products from previously unknown manufacturers or need to update brand information for accuracy.

The brand management tools integrate seamlessly with product management, enabling AI clients to establish proper product-brand relationships during food product registration and maintain data consistency across the nutrition database.

My MCP implementation includes sophisticated response formatting through the `PageResponse` class, which standardizes paginated data presentation across all tool methods. The `PageResponse.from(page)` pattern converts Spring Data `Page` objects into MCP-compatible responses that include both content and metadata.

This approach ensures consistent pagination handling across all tools, providing AI clients with predictable response structures that include total element counts, page information, and content arrays. The standardized pagination enables efficient large dataset handling while maintaining responsive API performance.

Default pagination handling throughout the tools ensures graceful operation when clients don't specify pagination parameters, automatically applying reasonable defaults that balance response size with data completeness.

My MCP tools integrate seamlessly with the existing service layer architecture, maintaining separation of concerns while providing AI access to core business functionality. Each MCP tool class delegates actual business logic to corresponding service classes (`ProductService` , `UserService` , `RecordService` , `BrandService`), ensuring consistent business rule enforcement regardless of access method.

This architectural approach means that MCP operations benefit from existing validation, security, business logic, and data persistence mechanisms without code duplication. The service layer abstraction also enables consistent behavior between MCP tool invocations, REST API calls, GraphQL operations, and gRPC requests.

Error handling within MCP tools leverages the underlying service layer's exception management, providing consistent error reporting and recovery mechanisms across all access patterns while maintaining the robustness and reliability expected from production nutrition tracking systems.

Conclusion

This project successfully demonstrates the implementation of a comprehensive AI-integrated nutrition tracking platform through dual architectural approaches. The **function calling client** delivers an intelligent conversational assistant with 8 specialized tools, multi-round execution capabilities, and robust security guardrails including prompt injection protection and rate limiting. Simultaneously, the **MCP server implementation** exposes 20+ standardized operations across 4 domain areas, enabling universal AI compatibility through the Model Context Protocol.

The technical implementation showcases production-ready architecture featuring reactive patterns, service layer integration, comprehensive error handling, and Docker deployment capabilities. The security-first design ensures reliable operation while maintaining scalability and performance standards.

In conclusion, this dual implementation strategy transforms a traditional nutrition management system into a modern AI-ready platform that supports both specialized conversational experiences and standardized AI integration pathways, providing maximum flexibility for diverse client needs while maintaining enterprise-grade reliability and security.

AI usage:

- this report
- help with understanding the task
- descriptions within annotations
- bugs searching
- help with prompt injection and token expenditure services (all used patterns were generated)
- help with understanding the concept of a function calling and how to implement it