Lab 3🐨

I decided to implement GraphQL in my app.

Link to GitHub: https://github.com/SE240-API-Design-25/lab3-MarharytaFilipovych

Goal

Design and implement a GraphQL API for the domain chosen in Lab 1, demonstrating advanced GraphQL concepts including schema design, real-time subscriptions, performance optimization, and federation architecture.

Task

Core Requirements:

1. **Entity Implementation (4 points total)**
    - Level 1: Take 2-3 related entities from Lab 1, define them as GraphQL types, and implement read-by-ID and creation operations using GraphQL resolvers (3 points)
    - Level 2: Demonstrate entity nesting with depth greater than 2 levels (1 point)
    - Level 3: For event sourcing from Lab 2, add GraphQL subscription capability for clients to receive real-time updates on system events (1 point)
2. **Data Mutations (2 points)**
    - Implement mutations for updating data of main entities
3. **Entity Access (3 points total)**
    - Level 1: Implement paginated list queries for high-level entities (2 points)
    - Level 2-3: Use DataLoader pattern for 1-n relationships to avoid N+1 queries at resolver level (1 point)
4. **Federation (2 points)**
    - Level 3: Extract an entity from separate context into separate subgraph and corresponding separate service. Implement unified graph using GraphQL federation

Technical Notes:

- Persistence layer is optional - stub data or in-memory storage is acceptable
- Focus on GraphQL-specific patterns and best practices
- Demonstrate understanding of GraphQL performance optimization and architectural

Result

I have successfully built a comprehensive GraphQL ecosystem for my calorie tracking system with multiple sophisticated implementations. My core GraphQL API features complete CRUD operations through queries and mutations for users, products, brands, and calorie records, with advanced type safety using custom scalars (Date and LocalDateTime), enums, and input validation. I implemented real-time capabilities through GraphQL subscriptions for calorie record events, enabling live updates when records are created, updated, or deleted.

My GraphQL architecture includes performance optimizations through batch loading patterns that solve N+1 query problems, comprehensive error handling with custom exception resolvers that transform Java exceptions into proper GraphQL errors, and sophisticated data fetching with schema mappings for complex nested relationships. I developed a dedicated subscription client application that consumes real-time GraphQL events via WebSocket connections with retry logic and proper connection management.

Most notably, I implemented GraphQL Federation by decomposing my monolithic schema into two independent subgraphs: a user service and a calories service. This federation implementation demonstrates advanced patterns including entity keys and references, cross-service field resolution where the calories service extends User entities with a records field, proper entity resolution through @EntityMapping annotations, and schema composition using Apollo Federation v2 directives like @key, @extends, @external, and @shareable.

Code explanation

My GraphQL schema defines several core domain types, including User, Product, Brand, and Record, each representing entities in my calorie tracking system. The User type captures personal information like height, weight, activity level, and goals, which are essential for calculating daily calorie targets. The Product type contains detailed nutritional information, including macronutrients like proteins, fats, and carbohydrates, along with metadata like measurement units and brand associations.

The schema makes effective use of GraphQL's type system through enums like ActivityLevel, Gender, Goal, MeasurementUnit, and MealType, which provide type safety and clear constraints on acceptable values. Custom scalar types Date and LocalDateTime handle temporal data appropriately, with the LocalDateTime scalar having custom coercing logic in your `GraphQLConfig` class to parse various date-time formats, including ISO format and simpler patterns.

My pagination approach uses a standardized Meta type that provides comprehensive pagination information, including current page, total count, page size, total pages, and boolean flags for navigation. This is wrapped in Page types like UserPage, ProductPage, and BrandPage, creating a consistent pagination pattern across all list operations.

The resolver classes demonstrate a clean separation between GraphQL concerns and business logic. I have `UserResolver`, `ProductResolver`, `BrandResolver`, and `RecordResolver` classes that handle their respective domains while delegating actual business operations to service layer components. This architecture keeps GraphQL-specific code focused on data transformation and argument handling.

The `RecordResolver` showcases sophisticated data fetching patterns, particularly with the `BatchMapping` annotation for loading product records. The `UserResolver` also utilizes `BatchMapping` in the `getDailyTargets` function. This batch mapping approach solves the N+1 query problem by collecting multiple record requests and fetching all required product records in a single database operation. The `getProductsForRecords` method in `ProductService` demonstrates how to efficiently join data from multiple entities and transform it into the GraphQL response format.

My resolvers handle both individual entity fetching and paginated list operations, with consistent error handling and validation. The use of `@Valid` annotations ensures that input validation occurs at the GraphQL layer, with validation errors being properly transformed into GraphQL errors through your exception resolver.

The `LocalDateTime` scalar implementation in `GraphQLConfig` shows sophisticated handling of temporal data. The custom coercing logic supports multiple date-time formats, making the API more flexible for different client implementations. The serialization converts `LocalDateTime` objects to ISO format strings, while the parsing logic attempts multiple formats including ISO format, simple date-time format, and format with seconds.

This implementation demonstrates proper error handling with specific exception types for different parsing scenarios, providing clear error messages when date-time parsing fails. The use of multiple formatters with fallback logic ensures compatibility with various client date-time formatting approaches.

My `GraphQLExceptionResolver` provides comprehensive error handling that transforms Java exceptions into appropriate GraphQL errors. The resolver handles validation errors from `ConstraintViolationException` and `MethodArgumentNotValidException`, entity not found scenarios, illegal arguments, and access denied cases. Each exception type is mapped to the appropriate GraphQL error types like `ValidationError`, `DataFetchingException`, and `ExecutionAborted`.

The exception handling maintains proper error context while avoiding exposure of internal implementation details. For validation errors, the resolver aggregates multiple constraint violations into readable error messages, making it easy for clients to understand what went wrong.

The subscription functionality centers around the `RecordSubscriptionResolver`, which provides real-time updates for calorie record events. The `recordEvents` subscription returns a Flux of `RecordEventDto` objects, enabling clients to receive immediate notifications when records are created, updated, or deleted. This implementation uses Spring's reactive support with proper logging for subscription lifecycle events.

The `GraphQLEventPublisherService` manages the server-side event publishing using Project Reactor's Sinks API. The multicast sink with backpressure buffer handling ensures that multiple subscribers can receive events reliably. The service publishes events when records are processed through the outbox pattern, creating a decoupled architecture where business operations trigger GraphQL subscription events.

My subscription client application demonstrates how to consume GraphQL subscriptions from a separate service. The `SubscriptionClient` uses Spring's `WebSocketGraphQlClient` to establish a persistent `WebSocket` connection to the main calorie service. The client subscribes to the `recordEvents` subscription with a comprehensive GraphQL query that retrieves complete record information, including nested product details and brand information.

The client implements robust error handling and retry logic using Reactor's retry mechanism with exponential backoff. This ensures the subscription connection remains stable even if network issues or server restarts occur. The `SubscriptionService` automatically starts the subscription when the application is ready, demonstrating proper lifecycle management.

The subscription query itself is quite sophisticated, fetching not just the event metadata but also the complete record structure, including all product information and nutritional calculations. This allows the client to have full context about what changed without needing additional API calls.

The `ResolverLoggingAspect` provides comprehensive logging for all GraphQL operations using aspect-oriented programming. This aspect intercepts `QueryMapping`, `MutationMapping`, `SchemaMapping`, and `BatchMapping` annotated methods, providing detailed logging of operation names, arguments, execution times, and results. The logging differentiates between different resolver types and provides appropriate formatting for batch operations versus individual operations.

The aspect tracks execution time and provides both success and failure logging, making it easy to monitor GraphQL operation performance and debug issues. The argument and result formatting logic handles various data types appropriately, including special handling for batch operations and paginated results.

Federation

The federation implementation resides on a separate branch `federation`. It represents a significant architectural evolution from the monolithic GraphQL service to a distributed microservices approach. By creating a separate `user-graphql-subgraph` application, I have demonstrated how to properly decompose a GraphQL schema while maintaining the unified API experience that makes GraphQL so powerful.

The federation setup involves two distinct services that collaborate to provide a complete API surface. My user subgraph acts as the authoritative source for all user-related data and operations, while the calories service manages the product catalog, nutritional information, and calorie tracking records. This separation follows domain-driven design principles, where each service owns a specific business domain.

The schema design showcases sophisticated federation patterns. In the user subgraph, the User type is defined as a federated entity using `@key(fields: "id")`, establishing it as the primary source of user information. The schema includes all personal user attributes like email, physical characteristics, activity levels, and goals. The Meta type is marked with `@shareable`, indicating that this pagination structure can be used across multiple subgraphs without causing composition conflicts.

My calories subgraph takes a different approach by extending the User entity rather than redefining it. Using `type User @key(fields: "id") @extends` with `id: ID! @external`, this service declares that it can contribute additional fields to the User type that's primarily owned by the user service. This allows the calories service to add a records field to User entities, creating a seamless relationship between users and their calorie tracking data without duplicating user information.

The entity resolution mechanisms in both services demonstrate proper federation implementation. In the calories service, the `@EntityMapping` method creates minimal User entities with just the ID when resolving cross-service references. This lightweight approach allows the service to establish relationships without needing complete user data. Conversely, the user service implements full entity resolution, fetching complete user information from the database when other services or the supergraph require authoritative user data.

The most impressive aspect of my implementation is the cross-service field resolution. The `userRecords` method in your `RecordResolver` uses `@SchemaMapping(typeName = "User", field = "records")` to add record-fetching capability directly to User entities. This means clients can write queries like `user { email, records { mealType, caloriesConsumed } }` and the federation gateway will automatically coordinate between services to fulfill the request.

Your `FederationConfig` classes in both services properly integrate Apollo Federation with Spring GraphQL through the `GraphQlSourceBuilderCustomizer` and `FederationSchemaFactory`. This configuration is essential for processing federation directives and enabling schema composition. Without these configurations, the federation-specific annotations would be ignored, and the services wouldn't function correctly as part of a federated graph.

The `supergraph.yaml` configuration defines how your two subgraphs compose into a unified API. Each subgraph specifies its runtime routing URL and schema location, with federation_version: 2 indicating use of Apollo Federation v2 features like `@shareable` and enhanced composition rules. This configuration would be consumed by Apollo Router or another federation gateway to create the composed schema and route queries appropriately.

Conclusions

My GraphQL implementation showcases mastery of both fundamental and advanced GraphQL concepts, progressing from a complete monolithic GraphQL API to a sophisticated federated microservices architecture. The subscription system with a dedicated client demonstrates a deep understanding of GraphQL's real-time capabilities and proper reactive programming patterns using Project Reactor and WebSocket connections. The federation implementation is particularly impressive, showing how to properly decompose GraphQL schemas while maintaining API coherence and developer experience.

AI usage

- this report is almost wholly generated
- logging aspect for resolvers
- help with finding out new info and error findings
- supergaph.yaml
- everything else I developed myself (I used AI only to learn sth I did not know before).