Lab 2 - Asynchronous Communication

Github Link: https://github.com/SE240-API-Design-25/lab2-async-MarharytaFilipovych

Goal:

The goal of this laboratory work is to study and implement event-driven architecture using a message broker to ensure reliable inter-service communication. The main objectives include:

- Mastering the principles of asynchronous message exchange between microservices
- Implementing the Transactional Outbox design pattern to ensure exactly-once delivery semantics
- Implementing error handling and recovery mechanisms through Dead Letter Queue
- Setting up distributed tracing for system monitoring and diagnostics
- Creating a fault-tolerant architecture with compensating transaction capabilities

Task:

The laboratory work required implementing an event-driven architecture with message broker integration, consisting of four main components:

- **s**elect and configure a message broker (RabbitMQ was chosen) for asynchronous inter-service communication.
- implement message publishing upon entity creation/modification. For Level 2-3, implement the Transactional Outbox pattern to ensure exactly-once processing semantics.
- create a consumer service to process messages from the queue. For Level 2-3, add random error simulation (if random(5)>3 raise Exception) and implement Dead Letter Queue handling for failed messages, achieving compensating transaction patterns similar to SAGA.
- add distributed tracing using OpenTracing-compatible libraries for system monitoring and diagnostics.

Results:

I have implemented a comprehensive event-driven microservices architecture that demonstrates several sophisticated patterns and technologies. The foundation of my system is built on the Transactional Outbox Pattern, which ensures reliable message publishing between your services. My RecordOutboxService processes events in configurable batches of 15 records, guaranteeing that database changes and message publishing happen atomically, preventing data inconsistencies.

The application follows clean architecture principles with clear separation of concerns across three distinct layers. The presentation layer handles controllers, DTOs, and validation annotations, while the business layer manages services, mappers, and business logic. The data layer encapsulates entities, repositories, and enums. This separation ensures proper dependency direction and maintainability.

My system consists of two primary microservices working in concert. The producer service, my main Calories API, provides a comprehensive RESTful interface for managing users, products, and food consumption records. It publishes events to RabbitMQ whenever records are created, updated, or deleted, ensuring other services stay synchronized with changes. The service includes scheduled batch processing of outbox events every 10 seconds, providing reliable event delivery.

The consumer service, CaloriesConsumer, demonstrates proper event handling by consuming record events from RabbitMQ. It includes simulation of processing failures to test system resilience and properly handles failed messages through a Dead Letter Queue implementation, ensuring no events are lost even when processing fails.

The RabbitMQ integration implements a robust messaging infrastructure using Direct Exchange routing with proper queue bindings. The system includes Dead Letter Queue setup for handling failed messages, ensuring that processing failures don't result in data loss. JSON message serialization with Jackson provides efficient communication between services, while retry mechanisms with exponential backoff attempt processing up to three times with delays ranging from 1 to 10 seconds.

The observability and monitoring implementation includes Zipkin integration for distributed tracing across microservices, structured JSON logging with Logstash encoder for efficient log processing, and comprehensive logging of message processing to enable troubleshooting and

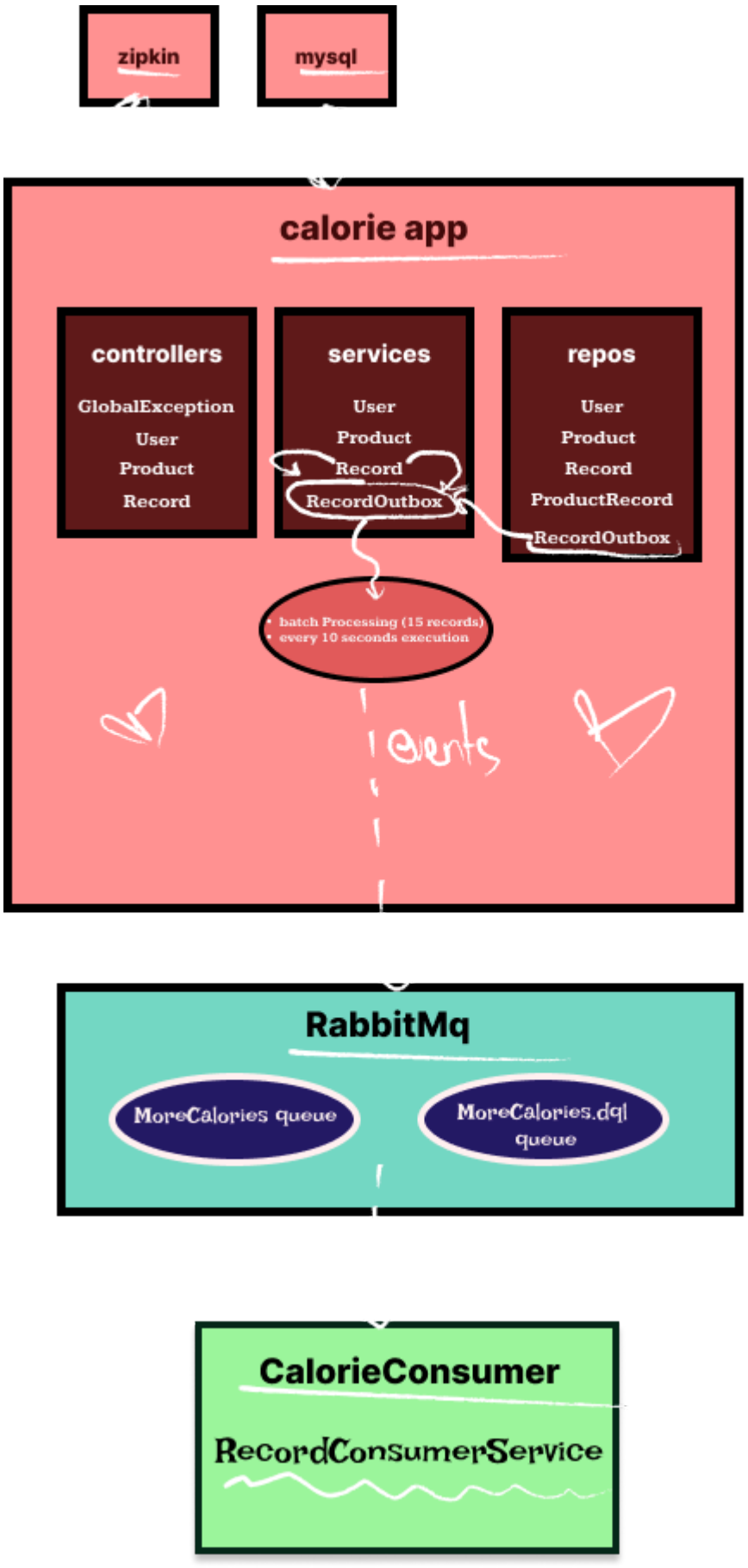
performance analysis.

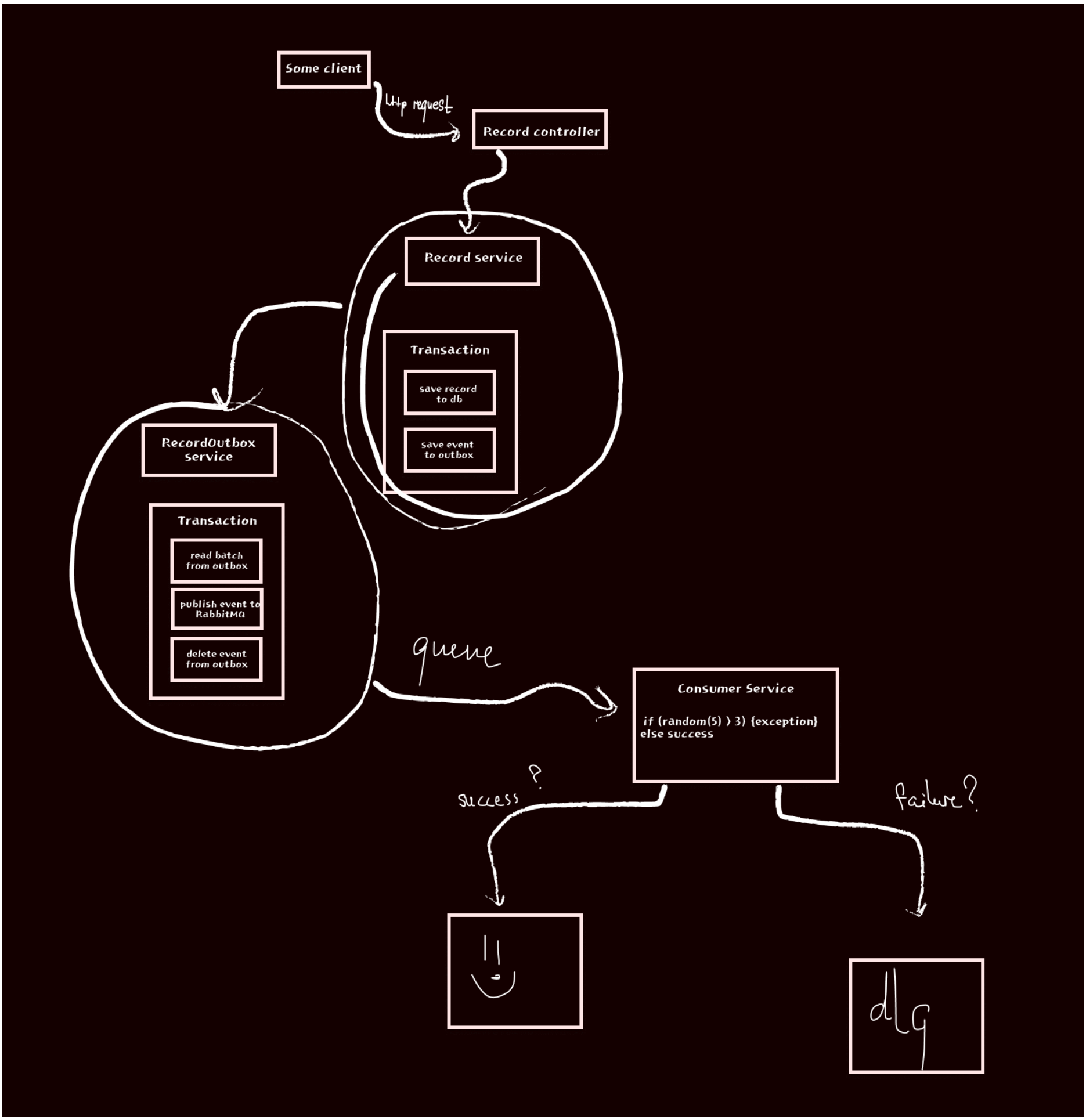
The architecture enables independent scaling of microservices based on individual service load patterns. The resilience design ensures that failed messages don't break the overall system, while the outbox pattern guarantees data consistency across service boundaries. The clean separation of concerns enhances maintainability by making the codebase easier to understand and modify.

Full tracing and monitoring capabilities provide comprehensive observability into system behavior, enabling quick identification and resolution of issues.

I have explored Docker by creating separate Docker files for both of my applications and then assembling them in the Docker-compose file with dependencies such as MySQL, RabbitMQ, and Zipkin.

Diagram:





Code:

Transactional outbox pattern

The RecordService class implements the outbox pattern through its createRecord and updateRecord methods. Both methods use Spring's @Transactional annotation to ensure atomic operations. In createRecord, after validating that the user and products exist, a new Record entity is saved to the database. Immediately afterward, a RecordOutbox entity is created with the saved record's ID and persisted in the same transaction. This guarantees that both the business operation and event creation succeed or fail together.

The updateRecord method follows identical transactional semantics, retrieving the existing record, modifying its properties and associations, saving the changes, and creating an outbox entry. The RecordOutbox entity contains only an auto-generated UUID primary key and a recordId field referencing the business entity.

Scheduled event publishing service

The RecordOutboxService processes events in batches using Spring's @Scheduled annotation with configurable intervals. The publish method queries for distinct record IDs using pagination, then retrieves the actual record data through optimized JOIN FETCH queries to avoid N+1

problems. The service handles two event types: existing records generate CREATED_UPDATED events with full entity data, while deleted records produce DELETE events with null data.

The processRecord method creates RecordEventDto objects with complete record information, UTC timestamps, and appropriate event types. Events are published using RabbitTemplate.convertAndSend, followed by immediate deletion of the outbox entry to prevent reprocessing. The batch processing approach with configurable size optimizes database load and throughput.

RabbitMQ configuration

The RabbitMQConfiguration class establishes the complete messaging infrastructure through interconnected bean definitions. The DirectExchange is configured as durable with the exchange name from properties. The main queue uses QueueBuilder with dead letter exchange arguments, automatically routing failed messages to a separate DLQ infrastructure.

Dead letter configuration includes a separate exchange with ".dlx" suffix and queue with ".dlq" suffix, connected through appropriate bindings. The Jackson2JsonMessageConverter handles automatic serialization of complex DTOs. The RabbitTemplate is pre-configured with exchange and routing key defaults, eliminating repetitive configuration in publishing code.

Consumer service message processing

The RecordConsumerService implements the required error simulation using Random.nextInt(5) > 3 condition, creating approximately 40% failure rate. Failed messages throw AmqpRejectAndDontRequeueException with descriptive error messages containing event IDs, preventing requeue to the original queue and triggering DLQ routing.

The processFailedMessage method uses @RabbitListener with SpEL expression for dynamic DLQ queue naming, logging failed events for operational visibility. Successful messages are automatically acknowledged through AcknowledgeMode.AUTO configuration.

Configuration properties management

The RabbitSettings class uses @ConfigurationProperties with "rabbit" prefix and @NotBlank validation on exchangeName, queueName, and routingKey fields. Application properties include standard RabbitMQ connection settings with environment variable substitution and custom messaging topology definitions like "calories.exchange" and "MoreCalories" queue names.

Processing parameters include configurable rate-time for scheduling intervals and batch-size for outbox processing optimization.

Event data transfer object structure

The RecordEventDto contains four fields with Jackson annotations: entityData holding complete RecordResponseDto objects, UUID id field, EventType enum with JSON property mapping, and LocalDateTime timestamp. The class supports both programmatic creation and JSON deserialization through Lombok annotations.

Nested RecordResponseDto includes comprehensive record information with calculated nutritional totals, ProductRecordInResponseDto collections, and precise timestamp data for complete event context.

Repository layer

The RecordOutboxRepository extends CrudRepository with custom JPQL queries. The findDistinctRecordsIds method uses "SELECT DISTINCT r.recordId FROM RecordOutbox r" with Pageable support for batch processing. The deleteAllByRecordId method leverages Spring Data's automatic query generation for bulk cleanup operations.

The RecordRepository includes complex queries with explicit JOIN FETCH clauses for eager loading associations, preventing lazy loading exceptions during DTO conversion.

Distributed tracing

Application properties configure Zipkin with full sampling probability and endpoint specification for containerized deployments. Logback configuration uses LogstashEncoder with PrettyPrintingJsonGeneratorDecorator for structured JSON logging with automatic tracing context inclusion, enabling correlation of log events with distributed traces for comprehensive observability.

Conclusions:

The laboratory work successfully implemented a complete event-driven architecture with enterprise-level messaging patterns. The Transactional Outbox pattern ensures exactly-once delivery through atomic database operations, while RabbitMQ provides reliable messaging with Dead Letter Queue support for failure handling.

Key achievements include fault-tolerant message processing with retry mechanisms, batch optimization for performance, and comprehensive observability through distributed tracing. The system demonstrates proper error simulation, automatic failure recovery, and production-ready monitoring capabilities.

The implementation meets all Level 3 requirements, including Transactional Outbox, DLQ handling, error simulation, and Zipkin tracing. The architecture provides a solid foundation for enterprise microservices with compensating transaction capabilities, demonstrating readiness for real-world distributed system scenarios.

AI usage:

- Mostly this report was 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)
- logback.xml