Relatório de Produção Individual - Parte 2

Sistema de Agendamento para Salão de Beleza

Guilherme G. Silvério

31/07/2025

1. Atribuição de Cargo e Tarefas

Nesta segunda e principal fase do projeto, os papéis exercidos foram expandidos para abranger todo o ciclo de vida do desenvolvimento de software, desde o backend distribuído até a interface do usuário e a garantia de qualidade.

- Arquiteto de Sistemas Distribuídos: Fui responsável por projetar e implementar a comunicação assíncrona entre a aplicação principal e o microserviço integrador, definindo as exchanges, filas e bindings no RabbitMQ para suportar os fluxos de dados bidirecionais.
- Desenvolvedor Full-Stack: Atuei tanto no backend, desenvolvendo a camada de persistência ODM com MongoDB e a API REST correspondente, quanto no frontend, criando a interface gráfica desktop com JavaFX e integrando-a diretamente ao contexto do Spring.
- Engenheiro de QA (Quality Assurance): Assumi o papel de garantir a qualidade e a corretude da lógica de negócio, escrevendo os testes unitários completos com JUnit e Mockito para as camadas de serviço ORM e ODM.

2. Contribuição de Acordo com a Atribuição

Esta fase foi focada em cumprir os "Trabalhos Futuros Pendentes" listados no relatório anterior.

O que foi cumprido:

1. Implementação da Camada ODM com MongoDB:
    - Foram criados os documentos denormalizados (AgendamentoODM) para servir como um modelo de leitura rápido e eficiente, seguindo o padrão CQRS.
    - Foi desenvolvida uma API REST completa com seu próprio Service e Controller para o gerenciamento administrativo direto desses documentos, demonstrando o desacoplamento da camada de leitura.

2. Desenvolvimento Completo do Middleware Integrador:
    - O middleware foi construído como um microserviço Spring Boot independente, responsável por orquestrar a sincronização de dados.
    - Fluxo ORM -> ODM (Denormalização): O listener que consome eventos do PostgreSQL foi finalizado. Ele busca os dados relacionais, os "achata" e os salva no MongoDB, criando o modelo de leitura.
    - Fluxo ODM -> ORM (Normalização): A lógica mais complexa do projeto foi implementada. Um listener consome eventos de agendamentos flexíveis criados no MongoDB, válida os dados "brutos" (como emails e nomes de serviços), busca as referências no PostgreSQL e monta as entidades relacionais, persistindo-as de forma transacional.
    - Fechamento do Ciclo: O middleware foi programado para, após salvar um dado normalizado no PostgreSQL, ele mesmo publicar um novo evento, garantindo que o modelo de leitura (AgendamentoODM) seja atualizado, fechando o ciclo de sincronização.
3. Criação da Interface Gráfica em JavaFX:
    - A aplicação JavaFX foi totalmente integrada ao contexto do Spring, permitindo a injeção de dependências (@Autowired) diretamente nos controllers da UI e eliminando a necessidade de um ApiClient HTTP.
    - Foi desenvolvida uma tela de login funcional que gerencia a sessão do usuário.
    - Foi criada uma tela principal com abas para demonstrar academicamente os dois fluxos:
        - Aba ORM: Um formulário estruturado que interage com o AgendamentoService do PostgreSQL.
        - Aba ODM: Um formulário flexível que demonstra a capacidade de um banco de dados de documentos de aceitar dados semi-estruturados, iniciando o fluxo de normalização.
4. Elaboração dos Testes Unitários:

- Foram criadas suítes de testes completas com JUnit 5 e Mockito para o AgendamentoService (ORM) e para o AgendamentoODMService (ODM).
- Os testes validam todos os métodos do CRUD, as regras de negócio (conflito de horário, antecedência de cancelamento) e os cenários de erro (IDs não encontrados, etc.), garantindo a robustez da lógica da aplicação.

3. Principais Dificuldades:

- Depuração Assíncrona: A maior dificuldade foi depurar o fluxo de mensagens entre as duas aplicações e o RabbitMQ. Erros como mensagens que não chegavam à fila ou loops de reprocessamento exigiram uma análise cuidadosa dos logs e da configuração do RabbitMQ em ambos os projetos.
- Lógica de Normalização: Projetar o OdmToOrmTransformerService foi complexo, pois exigiu a criação de uma lógica robusta para buscar entidades relacionais a partir de dados não-estruturados (como email e nomes de serviços), além de um tratamento de erros para não quebrar o listener em caso de dados inválidos.

4. Contribuição Além do Atribuído

O projeto evoluiu para uma implementação prática de padrões de arquitetura de software de nível profissional.

- Implementação Completa do Padrão CQRS: O sistema final é um exemplo funcional de Command Query Responsibility Segregation, com um modelo de dados otimizado para escrita (PostgreSQL) e outro otimizado para leitura (MongoDB).
- Arquitetura Orientada a Eventos: A comunicação entre os componentes foi feita de forma totalmente assíncrona, o que torna o sistema mais resiliente e escalável.