Projeto sobre Reactor Core/Web Reactive

Daniel Bravo*, Simão Sousa*
*Universidade de Coimbra, DEI
Emails: {bravo, simaosousa}@student.dei.uc.pt

***Abstract*—Este projeto explora o desenvolvimento de uma aplicação web reativa para a gestão de conteúdos multimédia e interações de utilizadores, abordando a necessidade de um sistema eficiente e responsivo. A questão central da investigação reside na complexidade da implementação de operações reativas em ambientes de dados dinâmicos. A dificuldade em garantir performance e escalabilidade na gestão de dados em tempo real torna este tópico relevante e desafiador. Para resolver este problema, foi utilizada uma abordagem baseada em Spring WebFlux e Reactor Core, permitindo a construção de um cliente e servidor altamente interativos. O método incluiu a implementação de serviços CRUD reativos e uma análise detalhada da performance, com ênfase nas otimizações das consultas do lado do cliente. Espera-se que os resultados deste estudo contribuam para melhores práticas no desenvolvimento de aplicações reativas, especialmente na gestão de dados multimédia.**

I. Introdução

Este relatório descreve o desenvolvimento e a implementação de uma aplicação web projetada para gerir conteúdos multimédia e interações de utilizadores, utilizando o modelo de programação reativa com Spring WebFlux. O projeto inclui um servidor que disponibiliza operações CRUD para dados de media e utilizadores, bem como uma aplicação cliente que consome esses serviços e gera relatórios. É feita uma análise da implementação e são destacados os desafios menos triviais, as soluções encontradas na elaboração das queries utilizadas no cliente, que otimizações foram implementadas e o seu impacto na performance. Por fim é feita uma análise das lições aprendidas, com foco em performance.

II. Ferramentas e Tecnologias

Para o desenvolvimento deste projeto, foram utilizadas as seguintes ferramentas e tecnologias:

- **Spring Boot [1] e Spring Framework [2]:** Base para a criação tanto do servidor como do cliente.
- **Reactor Core [3]:** Biblioteca utilizada para a programação reativa, proporcionando operadores como `Flux` e `Mono` para a manipulação de streams de dados.
- **Java 17 [4]:** Versão do Java utilizada.
- **PostgreSQL [5]:** Base de dados utilizada para o armazenamento persistente de dados de media e utilizadores.
- **Maven [6]:** Ferramenta utilizada para a gestão de dependências e estruturação do projeto.
- **Lombok [7]:** Utilizado para simplificar a escrita de código através de anotações que geram métodos como `getters`, `setters`, e `loggers`.
- **SLF4J [8]:** Interface de `logging` do servidor.
- **Docker [9] e Docker Compose [10]:** Empregados para criar um ambiente de desenvolvimento `containerizado`, facilitando a configuração e a replicação do ambiente de execução.
- **Postman [11]:** Ferramenta de `teste de API` utilizada para simular requisições e testar os endpoints desenvolvidos.

III. Arquitetura

A. *Servidor*

O servidor reativo desenvolvido expõe dados relacionados a media e utilizadores através de serviços web. As entidades **Media** e **User** são os elementos principais, cada um com os seguintes atributos:

- **Media**:
  - Identificador
  - Título
  - Data de lançamento
  - Classificação média (entre 0 e 10)
  - Tipo (Filme ou Série)
- **User**:
  - Identificador
  - Nome
  - Idade
  - Género

Para modelar a relação **many-to-many** entre **Media** e **User**, foi criada uma terceira tabela na base de dados que permite associar utilizadores a itens de media.

1) Limitações e Funcionalidades do Servidor: Este servidor é tratado como uma aplicação legacy com funcionalidades básicas, evitando-se melhorias que possam complicar as consultas do lado do cliente. As operações disponíveis são limitadas a operações CRUD simples, incluindo:

- Criação de media e utilizador
- Criação de relações entre media e utilizadores
- Leitura de todos os dados de media e utilizadores
- Leitura de dados específicos de media e utilizadores
- Atualização de media e utilizador específicos
- Eliminação de media e utilizador (apenas se não estiverem associados a outras entidades)
- Eliminação de relações
- Consulta de relações (retornando apenas os identificadores, sem dados detalhados)

Essa estrutura simplificada assegura que o servidor mantenha um foco em operações essenciais, evitando complexidade desnecessária e mantendo o desempenho adequado para integração com o cliente reativo.

B. Cliente

A arquitetura do cliente foi projetada para consumir os serviços web expostos pelo servidor de forma reativa, utilizando a biblioteca **Spring WebFlux** e a classe **WebClient** para a comunicação. A seguir, os principais componentes da arquitetura do cliente:

1) **WebClient**: É o principal componente responsável por realizar chamadas HTTP de forma reativa ao servidor. O cliente é instanciado com a URL base para simplificar as operações de consulta e manipulação de dados.
2) **Streams Reativos**:
   - Utilização de **Flux** para lidar com múltiplos itens de resposta do servidor (e.g., listas de objetos `Media` e `User`).
   - Utilização de **Mono** para tratar respostas que envolvem um único item ou resultado agregado.
3) **Operadores Reativos**:
   - **map**, **filter**, **flatMap** e **reduce** são amplamente usados para transformar, filtrar e processar os dados recebidos das chamadas ao servidor.
   - **retryWhen** é utilizado para gerir tentativas automáticas de novas chamadas em caso de falhas de rede, garantindo resiliência nas consultas.
4) **Relatórios**:
   - Os dados processados pelo cliente são estruturados em relatórios com a ajuda de objetos `StringBuilder`, o que facilita a concatenação de strings.
   - Os relatórios são criados com base em várias operações de consulta, como listagem de itens de media e contagens específicas, para serem gravados num ficheiro de texto.
5) **Gestão de Ficheiros**: O método `saveToFile` é responsável por gravar o conteúdo dos relatórios num ficheiro `.txt`.
6) **Gestão de Falhas de Rede**: Estratégias de resiliência incluem a utilização de operadores `retryWhen` e tratamento de erros com mensagens personalizadas, permitindo ao cliente continuar em execução mesmo em caso de falhas de rede.

IV. Desafios de Implementação do Cliente

A. Títulos e datas de lançamento de todos os itens de media

- **Descrição**: Esta query obtém todos os itens de media e extrai os seus títulos e datas de lançamento.
- **Desafio**: O principal desafio foi garantir que todos os itens fossem processados sem transformar o `Flux` em uma lista, mantendo a operação reativa.
- **Implementação**: Utilizou-se `bodyToFlux(Media.class)` para processar cada item de media e `doOnNext()` para adicionar os dados ao relatório.
- **Lição Aprendida**: Evitar operações que poderiam bloquear o fluxo assegurou que a execução permanecesse eficiente.

B. Contagem total de itens de media

- **Descrição**: Realiza uma contagem de todos os itens de media no servidor.
- **Desafio**: O uso de `count()` garantiu uma operação eficiente, mas foi necessário gerir o fluxo para que não houvesse bloqueios.
- **Implementação**: A contagem foi feita com `count()`, mantendo a natureza assíncrona do fluxo.
- **Lição Aprendida**: Operadores como `count()` facilitam a agregação de resultados em fluxos reativos sem prejudicar o desempenho.

C. Contagem de itens de media com classificação média $>$ 8

- **Descrição**: Filtra e conta os itens de media cuja classificação média é superior a 8.
- **Desafio**: Manter a eficiência no filtro e contagem sem interromper o fluxo.
- **Implementação**: Utilizou-se `filter()` para selecionar os itens com classificação $>$ 8 e `count()` para obter a contagem.
- **Lição Aprendida**: A aplicação de filtros em fluxos permite um processamento eficiente de grandes volumes de dados.

D. Contagem total de itens de media subscritos

- **Descrição**: Conta os itens de media que têm pelo menos um utilizador associado.
- **Desafio**: Verificar a lista de IDs de utilizadores para cada item e manter a execução não-bloqueante.
- **Implementação**: Foi usado `filter()` para identificar itens com listas de utilizadores não vazias e `count()` para obter a contagem.
- **Lição Aprendida**: O tratamento de listas aninhadas em fluxos pode ser feito de forma reativa com a utilização correta de operadores.

E. Itens dos anos 80 ordenados pela classificação média

- **Descrição**: Filtra itens lançados entre 1980 e 1989 e ordena-os pela classificação média.
- **Desafio**: A aplicação de filtros e ordenações reativas sem transformar o `Flux` em uma lista bloqueante.
- **Implementação**: `filter()` foi usado para limitar as datas de lançamento e `sort()` para ordenar pela classificação.
- **Lição Aprendida**: Manter o fluxo reativo mesmo em operações de ordenação requer cuidado para não perturbar a natureza assíncrona do processo.

F. Média e desvio padrão das classificações

- **Descrição**: Calcula a média e o desvio padrão das classificações dos itens de media.
- **Desafio**: Agregar dados de forma reativa sem bloquear o fluxo.
- **Implementação**: Para realizar este cálculo, foi utilizada a classe `Stats`, que implementa o algoritmo de Welford

para calcular a média e o desvio padrão de forma eficiente. A classe mantém o controle do número de valores (`count`), a média acumulada (`mean`) e a variância acumulada (`m2`). O método `addValue(double value)` permite adicionar cada classificação de forma incremental. Através do método `getStandardDeviation()`, o desvio padrão é calculado sem a necessidade de armazenar todas as classificações, evitando o uso de memória excessiva. Utilizou-se `map()` para extrair as classificações e `reduce()` para acumular os dados na instância da classe `Stats`.
- **Lição Aprendida**: Operações como `reduce()` podem ser usadas para agregar dados complexos em fluxos de forma eficiente. A utilização da classe `Stats` mostrou-se fundamental para manter a eficiência no cálculo estatístico sem comprometer a natureza reativa do fluxo.

G. Nome do item de media mais antigo

- **Descrição**: Encontra e retorna o item de media com a data de lançamento mais antiga.
- **Desafio**: Ordenar por data de lançamento e garantir que apenas o item mais antigo fosse processado.
- **Implementação**: `sort()` foi usado seguido de `next()` para pegar o primeiro item após a ordenação.
- **Lição Aprendida**: Utilizar `next()` após uma ordenação permite extrair o primeiro elemento de um fluxo de forma reativa.

H. Média de utilizadores por item de media

- **Descrição**: Calcula a média de utilizadores associados a cada item de media.
- **Desafio**: Contar utilizadores para cada item e calcular a média de forma eficiente.
- **Implementação**: `map()` contou os utilizadores e `reduce()` foi utilizado para calcular a média, utilizando a classe `Stats` para garantir um cálculo eficiente.
- **Lição Aprendida**: Operações que envolvem contagem e média podem ser feitas de forma não-bloqueante com operadores como `reduce()`.

I. Nome e número de utilizadores por item de media, ordenados por idade dos utilizadores

- **Descrição**: Retorna os nomes e o número de utilizadores por item de media, ordenados por idade em ordem decrescente.
- **Desafio**: Combinar múltiplos fluxos (media e utilizadores) e ordenar resultados de forma reativa.
- **Implementação**: `flatMap()` e `sort()` foram usados para processar e ordenar os utilizadores associados a cada item.
- **Lição Aprendida**: Operações que cruzam dados de diferentes fontes requerem uma coordenação cuidadosa com `flatMap()` para manter a reatividade.

J. Dados completos de todos os utilizadores, incluindo os nomes dos itens de media subscritos

- **Descrição**: Coleta os dados completos dos utilizadores e inclui os títulos dos itens de media subscritos.
- **Desafio**: Associar dados de media aos utilizadores sem comprometer a performance do fluxo.
- **Implementação**: `flatMap()` foi usado para ligar utilizadores a seus itens de media subscritos e `doOnNext()` para preparar os dados para o relatório.
- **Lição Aprendida**: Combinar fluxos de forma eficiente e preservar a performance foi essencial para lidar com grandes volumes de dados.

V. Lições aprendidas

Durante o desenvolvimento do cliente reativo para a aplicação de integração de sistemas, foram obtidas diversas lições importantes que contribuíram para o aprimoramento das competências em programação reativa e otimização de desempenho:

1) **Importância da Reatividade no Design de Aplicações**: A abordagem reativa mostrou-se essencial para criar um cliente capaz de lidar eficientemente com múltiplas chamadas de rede e processar fluxos de dados em tempo real. Com o uso de operadores como `Flux` e `Mono`, foi possível manter a natureza assíncrona e não bloqueante do programa, permitindo escalabilidade e melhor utilização dos recursos.
2) **Gestão de Concatenação e Paralelismo**: Aprendeu-se que a combinação de fluxos e o uso de operadores como `flatMap` e `zip` são fundamentais para manter a reatividade quando se trabalha com dados de diferentes fontes. Além disso, a utilização de operadores que permitem paralelizar certas operações foi crucial para reduzir latências e otimizar o tempo de execução.
3) **Tratamento de Erros Resiliente**: Implementar estratégias robustas de tratamento de erros, como `retryWhen` e `onErrorResume`, mostrou-se indispensável para garantir que o cliente permanecesse funcional mesmo em situações de falhas de rede ou respostas inesperadas do servidor. Isso reforçou a compreensão sobre como criar sistemas que toleram falhas e se recuperam sem comprometer a experiência do utilizador.
4) **Evitar Técnicas Bloqueantes**: O uso de operadores como `collectList()` foi evitado para impedir a perda da reatividade e a transformação de fluxos em operações bloqueantes. Esta prática ajudou a manter o desempenho do cliente e a escalabilidade das operações, sobretudo em consultas que manipulam grandes volumes de dados.
5) **Otimização de Performance**: Ao desenvolver consultas complexas e relatar os resultados em tempo hábil, foi importante aplicar otimizações que permitissem o processamento de dados em tempo real. O uso de operadores eficientes e a minimização de operações que bloqueassem o fluxo contribuíram para uma melhoria no desempenho geral da aplicação.

6) **Simplicidade na Criação de Relatórios**: Criar relatórios de forma incremental com o uso de `doOnNext()` permitiu um controle mais refinado sobre como os dados eram formatados e escritos. Esta abordagem ajudou a reduzir a carga de memória e a complexidade do código, mantendo a clareza e a facilidade de manutenção.
7) **Cooperação entre Múltiplos Fluxos de Dados**: A interação entre fluxos de diferentes endpoints, como a ligação de dados de media e utilizadores, destacou a importância de uma implementação bem planeada para que o cruzamento de dados fosse feito de forma eficiente. Operadores como `flatMap()` ajudaram a gerir múltiplas operações assíncronas sem introduzir latência excessiva.
8) **Manutenção da Reatividade com Operações de Ordenação e Filtros**: Utilizar operadores como `filter()` e `sort()` sem comprometer a natureza reativa do fluxo foi uma lição valiosa. A implementação cuidadosa desses operadores garantiu que a aplicação permanecesse responsiva e escalável, mesmo ao lidar com consultas mais complexas.

VI. Conclusão

O desenvolvimento desta aplicação web reativa com Spring WebFlux proporcionou uma compreensão aprofundada dos desafios e das melhores práticas na programação reativa. As lições aprendidas foram fundamentais para garantir um desempenho eficiente, mantendo a natureza não-bloqueante do fluxo de dados. O projeto demonstrou a eficácia do modelo reativo para aplicações que requerem operações de leitura e escrita rápidas e escaláveis, além de destacar a importância de estratégias de resiliência e de gestão de falhas.

Referências

[1] "Spring boot," https://spring.io/projects/spring-boot/, acedido: 5/11/2024.
[2] "Spring framework," https://spring.io/projects/spring-framework/, acedido: 5/11/2024.
[3] "Reactor core," https://projectreactor.io, acedido: 5/11/2024.
[4] "Java 17," https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html, acedido: 5/11/2024.
[5] "Postgresql," https://www.postgresql.org/, acedido: 5/11/2024.
[6] "Maven," https://maven.apache.org/, acedido: 5/11/2024.
[7] "Lombok," https://projectlombok.org/, acedido: 5/11/2024.
[8] "Slf4j," https://www.slf4j.org/, acedido: 5/11/2024.
[9] "Docker," https://www.docker.com/, acedido: 5/11/2024.
[10] "Docker compose," https://docs.docker.com/compose/ , acedido: 5/11/2024.
[11] "Postman," https://www.postman.com/product/what-is-postman/, acedido: 5/11/2024.