Universidade de Coimbra

Faculdade de Ciências e Tecnologia

Departamento de Engenharia Informática

Mestrado em Engenharia Informática

Integração de Sistemas

Reactor

Trabalho realizado por:

Miguel Ferreira | 2019214567 | PL1
Filipe Ribeiro | 2019223576 | PL1

1. Introdução

Neste projeto, foi proposto aos alunos que desenvolvessem uma aplicação Web que expusesse serviços e um cliente que consumisse esses mesmos serviços. Esta aplicação foi separada em duas partes: servidor e cliente. Quanto à parte do servidor, era esperada uma aplicação que mostrasse informação sobre alunos (id,nome,data de nascimento, créditos completos, média), professores (id, nome) e relações (student_id,professor_id), entre eles que suportasse operações CRUD (Create, Read, Update, Delete) simples. Quanto ao cliente, era esperado uma aplicação que recolhesse os dados do servidor e os mostrasse através de um ficheiro de texto.

2. Dados

Quanto aos dados, foi utilizada a base de dados fornecida pelo professor num container Docker onde criamos as diferentes tabelas que estão especificadas abaixo.

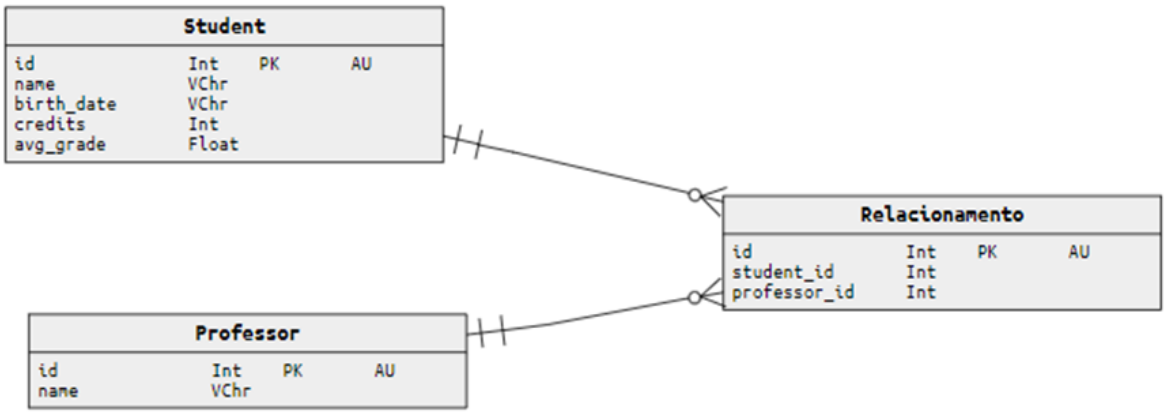

Figura 1- Diagrama Entidade Relacionamento

Observando o diagrama de entidade relacionamento da Figura 1, vemos que a base de dados da aplicação é constituída por 3 entidades:

- **Student -** é a entidade que representa todos os estudantes do sistema, onde o ID de cada um é a chave primária, que é incrementado automaticamente pela base de dados. Contêm outras informações acerca dos estudantes como o nome, data de nascimento, créditos realizados e média.

- **Professor -** representa todos os professores do sistema e como a entidade Student, o ID de cada um é a chave primária e é incrementado automaticamente. Além do ID a única informação presente é o nome do professor.

- **Relacionamento -** esta entidade é considerada fraca pois depende de duas outras (Student, Professor). Apresenta um ID para cada relação, o student_id e professor_id que é o id do aluno e do professor em cada relação.

Quanto à relação entre estudantes e relacionamentos, vemos que um estudante pode ter 0 ou vários relacionamentos e um relacionamento está apenas associado a um e um só estudante. Existe uma relação semelhante entre professores e relacionamentos, em que um professor pode ter 0 ou mais relacionamentos, mas um relacionamento está associado a um e um só professor.
Esta abordagem sobre a base de dados, foi uma tentativa de representar uma ligação Many-To-Many entre estudantes e professores.
Foi criado um código em SQL para facilitar o desenvolvimento da aplicação:

```sql
CREATE TABLE IF NOT EXISTS professor (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    FOREIGN KEY(id) REFERENCES relacionamento(professor_id) ON UPDATE CASCADE
);

--DROP TABLE student;
CREATE TABLE IF NOT EXISTS student (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    birth TEXT NOT NULL,
    credits integer NOT NULL,
    avg_grade float NOT NULL
    FOREIGN KEY(id) REFERENCES relacionamento(student_id) ON UPDATE CASCADE

);
--DROP TABLE relacionamento;
CREATE TABLE IF NOT EXISTS relacionamento (
    id SERIAL PRIMARY KEY,
    student_id integer NOT NULL,
    professor_id integer NOT NULL,
    FOREIGN KEY(professor_id) REFERENCES professor(id) ON UPDATE CASCADE,
    FOREIGN KEY(student_id) REFERENCES student(id) ON UPDATE CASCADE
);
```

Figura 2- Código de criação de tabelas da Base de Dados

Um ponto importante na criação das tabelas, é o facto de das variáveis ID das duas entidades serem enviadas como Foreign Key para a tabela de relacionamento, isto para não permitir criar relacionamentos com ID's inexistentes. O mesmo conceito se aplica na tabela de relacionamento, que são enviados os ID's dos professores e estudantes que têm uma relação como Foreign Key para permitir que não se use a operação de DELETE em objectos que têm uma relação.

3. Servidor

O servidor é uma aplicação legacy com funcionalidades básicas como Create, Read, Update e Delete.
A aplicação servidor está dividida em Entidades, Controladores, Repositórios e Serviços, em que cada secção está feita para gerir os dados da forma que se pretende.

- ## Reactivity

  Quanto à reatividade implementada na parte do servidor, foram criadas funções Flux e Mono para as diversas operações exigidas.
  Apenas foram criadas funções Flux quando é necessário reunir vários dados acerca de um objeto, como por exemplo na função getAllStudents() em que retorna todos os alunos presentes na base de dados. Esta função é comum para os 3 objetos (Students, Professors, Relacionamento).
  Para o resto das operações CRUD simples, foram utilizadas funções Mono.

- ## Data Access Layer

  A data access layer trata do processamento da informação que circula. Foram criados controladores para as diferentes entidades (Students, Professors, Relacionamento).
  Os controladores StudentControler.java, ProfessorController.java e RelacionamentoController.java suportados pelos Serviços e Repositórios, servem para realizar as operações de criação, atualização e destruição dos seus respectivos objetos nos endpoints pré-definidos.

- ## Exception Generation

  Quanto à geração de exceções, foram tratadas para todas as operações CRUD básicas.
  Um exemplo de um tratamento de uma exceção é na operação de criar, quer seja professor, estudante ou relacionamento, como podemos observar abaixo:

```java
@PostMapping(value = "/create")
@ResponseStatus(HttpStatus.CREATED)
public Mono<Relacionamento> createRelation(@RequestBody Relacionamento r) {
  return relacionamentoService.createRelation(r);
}
```

Figura 3- ResponseStatus

O @ResponseStatus indica se o pedido foi completado corretamente ou não. No caso de o pedido não for válido, da return da exceção.

4. Cliente

O cliente baseia-se no WebClient do Spring, com funções Reactive. Foi desenvolvida uma função para cada ponto do enunciado (getNamesAndBirthdates, getStudentNumber, getActiveStudents, getCoursesCompleted, getStudentsLastYear, getAvgGrade, getStdDev, getAvgGradeFinished, getStdDevFinished, getEldest, getAvgProfessorsPerStudent), com nomes autoexplicativos.

A cada uma das funções é fornecido o url que mostra todos os estudantes ("http://localhost:8080/student/all") e dentro de cada uma são realizadas as opções necessárias para obter os dados requeridos. Para funções específicas:

- Quando queremos restringir o tipo de estudantes devolvidos usamos *.filter*, que filtra os dados recebidos (similar a uma query de SQL)
- Quando queremos contar o número de elementos usamos *.count()*
- Para calcular uma média usamos o método de stream *.collect()* juntamente com o método *.averagingDouble* da classe *Collectors*
- Para calcular o desvio padrão foi usado novamente o método *.collect* juntamente com a classe *Collectors*, usando também a classe DoubleStatistics, que contém todas as funções necessátias para este cálculo
- Para obter o estudante mais velho, demos *.sort()* aos estudantes usando a data de nascimento como critério, e usando *.take(1)*, retiramos o estudante do topo.
- Para encontrar a média de professores por estudante, são obtidos dois Monos, um com o número de relações e outro com o número de estudantes. Estes monos são, entretanto, divididos um pelo outro de forma a obter a média pretendida.

Tomando especial atenção aos 3 últimos pedidos do cliente:

- Para o 9, é feita uma contagem, do número de relações através do *.count()*, e guardado num Mono<Long> e, de seguida é feita uma contagem do número de estudantes, também através do *.count()* e guardada noutro Mono<Long>. Estes dois valores são, de seguida, divididos um pelo outro através do Mono.zip de forma a retornar a média de professores por cada estudante.
- Para o 10, percorremos a tabela de Students através de um flatMap e, para cada Student, percorremos a tabela de relações. Por sua vez, cada entrada da tabela de relações percorre a tabela de professores de forma a encontrar o professor associado àquele estudante. Para calcular o número de estudantes por professor, por cada entrada da tabela de professores verificamos quantas relações existem, apresentando esse número com o *.count()*.
- Em relação ao 11, é muito similar ao 10. Percorremos a tabela de estudantes, seguida da de relações, seguida da de professores. Caso o estudante tenha professores associados, as suas informações vão ser apresentadas, seguidas do nome do professor. Caso o Student não tenha professores associados, é imprimida a sua informação seguida de uma mensagem a informar que não tem professores. esta mudança é feita através do método *.switchIfEmpty()*.

Em relação a funções auxiliares, a *getWebClient* foi criada para inicializar o WebClient e a *writeToFile* para enviar o resultado de cada função reativa para um ficheiro de texto.

O output do cliente é escrito para um ficheiro de texto *output.txt* com a legenda apropriada.

5. Problema de otimização

A performance da parte do cliente não é ótima, visto que fazemos o mesmo pedido à API várias vezes (lista de todos os students) e esta questão poderia ter sido otimizada, fazendo apenas o pedido uma vez e tratando todas as questões relacionadas com aquele conjunto de dados de uma vez.

6. Referências

[1] - JAVA Streams - Standard Deviation. [*S. l.*], 2019. Disponível em: https://stackoverflow.com/questions/36263352/java-streams-standard-deviation. Acesso em: 1 nov. 2022.

[2] - REACTOR 3 Reference Guide. [*S. l.*], 2022. Disponível em: https://projectreactor.io/docs/core/release/reference/. Acesso em: 2 nov. 2022.

[3] - SPRING Boot R2DBC Example. [*S. l.*], 2022. Disponível em: https://github.com/kamalhm/spring-boot-r2dbc. Acesso em: 3 nov. 2022.

[4] - GUIDE to Spring 5 WebFlux. [*S. l.*], 2022. Disponível em: https://www.baeldung.com/spring-webflux. Acesso em: 4 nov. 2022.