



• U • C •

FCTUC FACULDADE DE CIÊNCIAS  
E TECNOLOGIA  
UNIVERSIDADE DE COIMBRA

INTEGRAÇÃO DE SISTEMAS

# Relatório - Trabalho 1

Bruno Baptista nº2018278008  
& Simão Vazão nº2018289426

PL2

11 de novembro de 2022

## Conteúdo

1	Introdução	2
2	Base de dados	2
3	Implementação do servidor	2
3.1	Repositórios	3
3.2	Controladores	3
3.3	Serviços	3
4	Implementação do cliente	3
4.1	Requisitos 1	3
4.2	Requisito 2	3
4.3	Requisito 3	3
4.4	Requisito 4	3
4.5	Requisito 5	4
4.6	Requisito 6	4
4.7	Requisito 7	4
4.8	Requisito 8	4
4.9	Requisito 9	4
4.10	Requisito 10	4
4.11	Requisito 11	4
4.12	Otimização de queries	5
4.13	Outros detalhes	5
4.13.1	Retry	5
4.13.2	Logging	6
5	Conclusão	6

# 1 Introdução

Programação reativa é um paradigma de programação que utiliza fluxos de dados e propagação de estados.

Os fluxos de dados deste modelo de programação, caracterizam-se por serem em grande parte, assíncronos e não bloqueantes.

Estas características trazem vantagens relativamente à forma tradicional imperativa de programação, uma vez que a transmissão de dados assíncronos para um consumidor(Subscriber), permite que o envio seja feito de imediato, à medida que a informação fica disponível por parte do servidor(Publisher), o que possibilita o desenvolvimento de programas que respondam de forma rápida e assíncrona.

Consequentemente de forma a abordar este tema foi desenvolvido um projeto em Reactor Flux, seguindo o modelo padrão da estrutura de aplicações Spring Controller-Service-Repository.

# 2 Base de dados

Uma vez que os dados a serem utilizados são estudantes, professores e as relações entre eles, foi criada uma base de dados, utilizando PostgreSQL 14, com as seguintes tabelas:

students	
PK	<u>id SERIAL</u>
	name VARCHAR(255) birthday VARCHAR(255) credits INT avg_grade FLOAT

(a) Tabela dos estudantes

professors	
PK	<u>id SERIAL</u>
	name VARCHAR(255)

(b) Tabela dos professores

relationships	
PK	<u>id SERIAL NOT NULL</u>
	s_id BIGINT NOT NULL p_id BIGINT NOT NULL UNIQUE (s_id, p_id)

Figura 2: Tabela das relações

Os id das varias entidades são utilizados como Primary Keys, e fazem proveito da propriedade *SERIAL*, para que seja gerado um id novo na inserção na base de dados, não tendo o servidor de gerar um id.

Na tabela de relações optamos por utilizar a propriedade *UNIQUE* para as combinações entre id de estudante e id de professor, de modo a não repetir varias vezes a mesma relação.

# 3 Implementação do servidor

Foi desenvolvido um servidor reativo Spring, utilizando Spring Webflux para lidar com streams reativas, e Spring Boot R2DBC, para ligar o servidor a base de dados.

O servidor tem vários *endpoints* que permitem a um cliente realizar simples operações CRUD, através de pedidos REST. Ao aceder a estes *endpoints*, o servidor retorna a um cliente a informação pedida através de streams do Webflux, em formato JSON.

Nas seguintes subsecções entramos em detalhe sobre as varias componentes do servidor reativo implementado, nomeadamente os repositórios, controladores e serviços.

### 3.1 Repositórios

Os repositórios são as classes que usamos para ir buscar informações à base de dados, temos três classes diferentes com um repositório, Professors, Students e relationships.

Nos repositórios ao estender a classe `ReactiveCrudRepository<Student, Long>` temos acesso a métodos pre definidos que nos permitem realizar as operações CRUD, no entanto para a realização da operação `ReadRelationship` foram criadas queries personalizadas no repositório da classe `relationships`.

Foi criada uma query, que, recebendo um id de um estudante, retorna os ids dos professores associados a este, e outra query, que, recebendo um id de um professor, retorna os ids dos estudantes associados a este.

### 3.2 Controladores

Os controladores são as classes que usamos para receber as requests via REST do cliente, analisamos as requests através do endpoint especificado e chamamos métodos da classe dos serviços.

Temos três classes com controllers, Students, Professors e Relationships.

### 3.3 Serviços

Os serviços servem de intermediário entre os controllers e os repositórios. Desta forma temos serviços de três classes diferentes, Professors, Students e Relationships.

## 4 Implementação do cliente

Os primeiros 8 requisitos são bastante semelhantes, no sentido que em todos são feitos requests ao servidor para obter todos os estudantes, e são depois realizadas operações sob o fluxo de estudantes recebidos.

Já os requisitos 9-11 são mais complicados, pelo que requerem múltiplos fluxos de informação, provenientes de requests ao servidor para obter estudantes, professores e relações.

#### 4.1 Requisitos 1

Para o requisito 1, é nos pedido o nome e a data de nascimento de cada aluno. Para isso, após a receção do fluxo, apenas é feito um map sobre estudantes recebidos, onde se obtém o nome e a data de nascimento de cada estudante.

#### 4.2 Requisito 2

No requisito 2, apenas nos é pedido o número de estudantes, pelo que, foi realizada uma contagem através da função `count()` após a receção dos estudantes no fluxo.

#### 4.3 Requisito 3

Este requisito tem um funcionamento parecido com o requisito 2, mas neste só queremos o número de estudantes ativos, antes de aplicar o `count()`, filtramos os estudantes apenas com menos de 180 créditos, através da função `filter()`.

#### 4.4 Requisito 4

Para o requisito 4, procuramos saber o número de disciplinas completadas por cada estudante. Este requisito tem um funcionamento parecido com o requisito 1, só que no map é calculado o numero de disciplinas completas, através do numero de créditos que cada estudante tem.

## 4.5 Requisito 5

Para obter a informação de todos os estudantes no último ano de graduação, no requisito 5, filtramos os alunos que se encontram no último ano, ou seja que tem menos que 180 créditos, e pelo menos 120 créditos, através de um *filter()*. Uma vez que nos é pedida esta informação ordenada por proximidade de conclusão, recorremos à função *sort()* no fluxo filtrado, ordenando por numero de créditos.

## 4.6 Requisito 6

Neste requisito queremos a média e o desvio padrão das notas de todos os estudantes. Para tal, utilizamos um *map()* para obter a nota média de cada estudante recebido através do fluxo, e utilizamos a função *collectList()* para receber todas as notas de uma vez, transformando o fluxo em mono.

Após o collect das notas, através de um *flatMap()* calculamos a média e o desvio padrão.

## 4.7 Requisito 7

O requisito 7 é igual ao requisito 6, mas só procuramos saber a média e o desvio padrão das notas de alunos graduados. Para tal, após receber os estudantes através do fluxo, usamos um *filter()* para filtrar apenas os alunos com 180 créditos.

## 4.8 Requisito 8

No requisito 8 queremos obter o nome do estudante mais velho. Para isso, através de um *sort()*, ordenamos os alunos recebidos no fluxo por idades, e de seguida, através da função *elementAt()*, acedemos ao primeiro elemento no fluxo, que neste caso vai ser o aluno mais velho.

## 4.9 Requisito 9

Para obter o número médio de professores por estudante, no requisito 9, tivemos de usar vários fluxos.

O primeiro fluxo é utilizado para obter o número de estudantes, através de um *count()*, para ser utilizado no cálculo da média.

O segundo fluxo tem como objetivo obter os ids dos estudantes. Estes ids vão ser utilizados na request ao servidor para obter o fluxo de ids de professores a que cada estudante está associado, sendo feito um *count()*.

Como queremos contar o número de professores associados a cada estudante, fazemos então um *reduce()*, para somar todos os *count()* anteriores

## 4.10 Requisito 10

Neste requisito, é nos pedido o nome e número de todos os estudantes por professor. Para tal recorremos a criação de uma classe, composta por um objeto do tipo Professor, e um ArrayList para guardar os nomes de cada estudante associado ao professor.

É feita uma request ao servidor para obter um fluxo com todos os professores, e cada um destes professores é mapeado para a classe composta referida previamente, através de um *map()*.

De seguida, através do id do professor, é pedido ao servidor um fluxo com os ids dos estudantes associados ao professor. Através de cada um destes ids, é feita novamente uma request ao servidor para obter o estudante associado a este id, obtemos o seu nome. Os nomes dos estudantes são então adicionados a uma ArrayList contida na classe composta, através da função *doOnNext*. Por ultimo, uma vez que queremos os professores ordenados por numero de estudantes, é feito um *sort* para os ordenar.

## 4.11 Requisito 11

No requisito 11, queremos a informação completa de todos os estudantes, adicionando o nome dos seus professores.

Semelhante ao requisito 10, recorremos à criação de uma classe auxiliar, composta por um estudante e uma ArrayList com os nomes dos professores associados.

Fazemos uma request ao servidor para obter um fluxo com todos os estudantes, e mapeamos os estudantes que vamos recebendo para a classe composta, tal como para o requisito 10.

De seguida, através do id do estudante, pedimos ao servidor um fluxo com os ids dos professores associados a cada estudante. Os ids são obtidos através da tabela de relações. Após isto, para cada um destes ids recebidos, pedimos ao servidor o nome professor a que este id pertence.

Com a função *doOnNext* adicionamos os nomes obtidos a ArrayList contida na classe composta.

## 4.12 Otimização de queries

Inicialmente optámos por realizar uma request por requisito, para posteriormente comparar a velocidade de execução dos requisitos e a sua respetiva escrita para o ficheiro.

Para os testes de otimização foram criados 400 dados de professores, 400 dados de estudantes e 127 relações entre professores e estudantes.

Os testes foram realizados dentro de um ciclo de 30 iterações e o tempo médio foi calculado. Numa tentativa de otimização tentámos usar os métodos PublishOn e SubscribeOn com os respetivos Schedulers, para paralelizar as operações do pipeline do fluxo, no entanto as velocidades de execução dos requisitos com ou sem estes métodos revelaram-se idênticas, sendo que demoravam 3 segundos para ambas as execuções(2888ms sem Schedulers e 2678ms com schedulers).

Tentámos ainda unir os pipelines de vários fluxos sob a mesma request via Rest, mas os tempos destas operações revelaram-se maiores do que em requests separadas(cerca de 4 segundos por execução).

```
for(int i=0;i<30;i++) {  
  
    long start = System.currentTimeMillis();  
  
    Thread T1=new Thread () -> client.getStudentNamesBirthdays(file1).blockLast();  
    Thread T2= new Thread () -> Flux.from(client.getNumberStudents(file2)).blockLast();  
    Thread T3= new Thread () -> Flux.from(client.getActiveStudents(file3)).blockLast();  
    Thread T4= new Thread () -> client.getCompletedCourses(file4).blockLast();  
    Thread T5= new Thread () -> client.getLastYearStudents(file5).blockLast();  
    Thread T6= new Thread () -> Flux.from(client.getAvgStdGrades(file6)).blockLast();  
    Thread T7= new Thread () -> Flux.from(client.getAvgStdFinished(file7)).blockLast();  
    Thread T8= new Thread () -> Flux.from(client.getEldestStudent(file8)).blockLast();  
    Thread T9= new Thread () -> Flux.from(client.avgProfs(file9)).blockLast();  
    Thread T10= new Thread () -> client.getStudentsPerProfessor(file10).blockLast();  
    Thread T11=new Thread () -> client.getStudentsWithProfessors(file11).blockLast();  
  
    T1.start();T2.start();T3.start();T4.start();T5.start();T6.start();T7.start();T8.start();T9.start();T10.start();T11.start();  
    T1.join();T2.join();T3.join();T4.join();T5.join();T6.join();T7.join();T8.join();T9.join();T10.join();T11.join();  
}
```

Figura 3: Exemplo utilização de threads

```
public Mono<Long> getActiveStudents(File file) {  
    return this.webClient.get().uri("api/students") capture of ?  
        .retrieve() ResponseSpec  
        .bodyToFlux(Student.class).publishOn(Schedulers.boundedElastic()) Flux<Student>  
        .filter(student -> student.getCredits() < 180) // filtra os estudantes que têm menos de 180 credits  
        .count().publishOn(Schedulers.boundedElastic()) Mono<Long>  
        .map(total -> {  
            writeToFile(file, Content: total + " active Students");  
            return total;  
        });  
}
```

Figura 4: Exemplo PublishOn

## 4.13 Outros detalhes

### 4.13.1 Retry

Foi criada uma função do lado do cliente com o intuito de mostrar a tolerância à falha de rede. Para tal, utilizamos a função *retryWhen*, que nos permite tentar restabelecer ligação com o servidor 3 vezes.

Caso a ligação ao servidor não seja restabelecida, utilizamos a função *onErrorResume*, onde desligamos o cliente.

```

public Flux<Student> testRetry() {
    return this.webClient.get().uri(uri: "api/students") capture of ?
        .retrieve() ResponseSpec
        .bodyToFlux(Student.class) Flux<Student>
        .retryWhen(Retry.backoff( maxAttempts: 3, Duration.ofSeconds(1))
            .doAfterRetry(retrySignal -> {
                System.out.println("Retried " + retrySignal.totalRetries());
            }))
        .onErrorResume(throwable -> {
            System.out.println("Server is off!");
            System.exit( status: -1);
            return Flux.empty();
        })
        .doOnComplete() -> System.out.println("Connected to server!");
}

```

Figura 5: Função para restabelecer ligação ao servidor

#### 4.13.2 Logging

Para implementar o logging do lado do servidor, bastou importar a dependência *log4j2* no maven, e a sua configuração é feita através do ficheiro *log4j2.xml*. O logging do servidor é guardado no ficheiro *server.log*, onde se pode consultar todas as informações relacionados com a execução do servidor.

```

<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="INFO">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n" />
        </Console>
        <File name="MyFile" fileName="server.log" immediateFlush="false" append="false">
            <PatternLayout pattern="%d{yyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
        </File>
    </Appenders>
    <Loggers>
        <Root level="debug">
            <AppenderRef ref="Console" />
            <AppenderRef ref="MyFile" />
        </Root>
    </Loggers>
</Configuration>

```

Figura 6: Ficheiro de configurações do log4j2

## 5 Conclusão

Através da implementação de um servidor e cliente reativos, conseguimos perceber várias vantagens de utilizar programação reativa.

A maior vantagem é o facto de um servidor não necessitar de enviar os dados todos simultaneamente, permitindo que um cliente seja servido progressivamente. Isto leva a que um cliente não necessite de esperar que o servidor prepare os dados todos, e possa começar a tratar dos dados imediatamente.

Permitiu-nos ainda trabalhar sobre questões relacionadas com a performance e tratamentos de exceções e como esta pode ser feita, apesar de no nosso caso, a performance dos tempos não ter sido possível.