UNIVERSIDADE D
COIMBRA

Gui Costa, 2021186342
Dário Félix, 2018275530

Trabalho 2
SPRING REACTIVE

Relatório no âmbito da cadeira de Integração de Sistemas do Mestrado em Engenharia Informática, orientado pelo Professor Filipe Araújo (PL1), do Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologias da Universidade de Coimbra.

11 de novembro de 2022

Conteúdo

Índice **I**

1 Introdução **1**

2 Implementação & Discussão **1**
2.1 *Reactive Server* 1
2.1.1 Gerador de Dados 1
2.1.2 Emular Atrasos de Rede 2
2.1.3 Tolerar Falhas na Rede 2
2.2 *Client* 2
2.2.1 Optimização da Aplicação 2
2.2.2 Efeito dos Atrasos de Rede 3
2.2.3 Tolerar Falhas na Rede 4

3 Conclusão **4**

4 Referências **4**

1 Introdução

Este trabalho tem como objetivo aprender a programar usando um modelo de programação reativa.

Para tal, desenvolveu-se uma aplicação web que permitisse acesso a determinados serviços e uma aplicação cliente que fizesse acesso a esses serviços. Assim, utilizou-se *Project Reactor* que oferece duas interfaces, *Flux* e *Mono* que respetivamente aceita N sequencias ou 0-1 sequencias. Estão também preparadas para *backpressure* e são *non-blocking*.

2 Implementação & Discussão

2.1 *Reactive Server*

No caso da aplicação *web*, os serviços fornecidos limitam-se a operações de CRUD, tendo como partida o "Spring Boot R2DBC Example" [4], de acordo com os dados definidos na lista que se segue:

Estudante:

1. Identificador
2. Nome
3. Data de aniversario
4. Créditos completos
5. Media das notas obtidas.

Professor:

1. Identificador
2. Nome

2.1.1 Gerador de Dados

Foi criado um gerador [3] com o objetivo de gerar dados aleatórios e popular a base de dados. O gerador aceita números diferentes de professores, estudantes e professores por estudante. O gerador esta ligado ao repositório de cada das tabelas (*Professor, Student e ProfessorStudent*), para obter acesso ao repositório o gerador usa uma classe auxiliar que obtém o contexto da aplicação e retorna o *bean* pedido.

Para preencher as colunas das tabelas *Student* e *Professor*, o gerador gera nomes, datas, créditos e medias aleatórias através da livrarias *JavaFaker*. A geração de alunos

e professores é feita em paralelo, ou seja, existem duas *threads*, uma que gera alunos e outra que gera professores. Após terminarem as *threads* são geradas as relações.

É de salientar que o gerador devido a usar os repositórios também usa *Flux* e os seus métodos.

2.1.2 Emular Atrasos de Rede

De forma a emular os atrasos de rede e avaliar o seu impacto na execução da aplicação do cliente, implementou-se uma classe para este feito conforme o Código 2.1. A cada *query* é gerado um número aleatório dentro dos limites impostos que corresponderá à duração do atraso. Antes de inicializar a *Spring Application*, é possível ativa-lo/desativa-lo, definir os seus limites inferior e superior e a *seed*.

```
// ServerApplication - Main:
EmulateNetworkDelays.setActive(true);
EmulateNetworkDelays.setSeed(0);
EmulateNetworkDelays.setMinDelayMillis(0);
EmulateNetworkDelays.setMaxDelayMillis(50);

// Exemplo controller:
this.studentRepository.findAll().delaySequence(EmulateNetworkDelays.
    emulate());
```

Código 2.1: Exemplo de emulação de atrasos na rede.

2.1.3 Tolerar Falhas na Rede

Para testar eventuais falhas do servidor ou da rede, foi criado um serviço que recusa as primeiras duas tentativas de pedido ao servidor, emitindo uma exceção com o *HTTP Status Code* 500 (*Internal server error*). Após as duas tentativas funciona como um serviço normal e retorna os dados das relações ao cliente. O erro retornado e simplesmente um fluxo que retorna erro imediatamente após o *subscribe*.

2.2 *Client*

2.2.1 Optimização da Aplicação

Do ponto de vista do cliente, tentou-se otimizar ao máximo as *queries*, por um lado reduzindo em número (fazer o mesmo em menos *queries*), e por outro, tentar executá-las em paralelo (apartir de *threads* implícitas).

Por exemplo, tanto o requisito 10 como o requisito 11, causam algum *overhead*, pois a cada elemento recebido gera novas *queries*, e como não utilizam a mesma *query*, concluiu-se que seria vantajoso executá-las em paralelo, a começar antes de todos os outros requisitos, e sem bloquear o programa (pela utilização das *threads*). Assim,

utilizou-se semáforos, não só para esperar mais tarde pela sua conclusão, como também no controlo das precedências de requisitos, devido à existência de dependências de resultados intermédios, Código 2.2.

```
CountDownLatch semaforoReq10 = new CountDownLatch(1);

// REQ 10:
this.getAllProfessor()
    // ...
    .doFinally(signalType -> semaforoReq10.countDown())
    .subscribe();

// Continuar a execucao do resto da aplicacao sem bloquear
// ...

// Esperar pela conclusao do REQ 10
try {
    semaforoReq10.await();
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Código 2.2: Exemplo da utilização de semáforos.

Em relação aos primeiros 9 requisitos, foi-se progressivamente diminuindo o número de *queries* e a sucessiva manutenção dos mesmos, até ter apenas uma *query*, Código 2.3. De facto, e devido à natureza simples destes requisitos, bastou a utilização de variáveis temporárias para o cálculo de somatórios, contadores de elementos, etc.

```
this.getAllStudents()
    // REQ 1:
    .doOnNext()

    // REQ 2:
    .doOnNext()

    // ...

    // REQ N:
    .doOnNext()

    .count()
    .block();
```

Código 2.3: Exemplo da utilização da mesma query para vários requisitos.

2.2.2 Efeito dos Atrasos de Rede

Efetivamente, o efeito dos atrasos na rede é bastante notório nos requisitos 10 e 11 devido àquilo que já foi referido: a cada elemento da *query* inicial geram-se outras novas *queries* em complexidades na ordem de $\boldsymbol{O(E \cdot R) \cdot delay}$ para o requisito 11 e $\boldsymbol{O(P) \cdot delay}$ para o requisito 10, acrescidos do *delay*, onde **E** representa a quantidade de estudantes na base de dados, **R** representa a quantidade de relações que cada estudante tem, e **P** representa a quantidade de professores.

2.2.3 Tolerar Falhas na Rede

Para tolerar falhas de rede ou do servidor foi implementa uma *query* que tenta-se conectar ao servidor mais do que uma vez, caso falhe ele espera 5 segundos e tenta a conectar de novo, o tempo entre cada tentativa é exponencial e chamado de *backoff*. No total são feitas 3 tentativas. A cada tentativa e medido o tempo que demorou desde o inicio das tentativas bem como a exceção gerada pelo servidor. Caso o servidor acabe por responder funciona normalmente. Nesta *query* é feito o uso dos métodos *retryWhen*, *doAfterRetry* e *Retry.backoff* das classes *Flux* e *Retry* respetivamente [5].

3 Conclusão

Os objetivos foram cumpridos. As maiores dificuldades centram-se na otimização das *queries* de forma minimizar o numero de pedidos ao servidor e a sua velocidade, bem como gerir os recursos a que as *threads* geradas automaticamente tem acesso.

Esta forma de programar tem vantagens, o código e *non-blocking* o que significa que a aplicação não tem de esperar que o pedido acabe para executar outras tarefas tornando-as muito mais eficientes, no entanto isto implica uma boa gestão dos tempos e de como se utiliza os dados, requerendo algum conhecimento sobre *Threads*, *Scheduling* e assincronia.

4 Referências

[1] Wikipedia, "Reactive programming". Disponível em `https://en.wikipedia.org/wiki/Reactive_programming`

[2] Project Reactor, "Create Efficcient Reactive Systems". Disponível em `https://projectreactor.io/`

[3] Java Faker, "Generate Fake data". Disponível em `https://github.com/DiUS/java-faker`

[4] Spring Boot R2DBC Example, "An example implementation of Spring Boot R2DBC REST API with PostgreSQL database". Disponível em `https://github.com/kamalhm/spring-boot-r2dbc`

[5] Baeldung, "Guide to Retry in Spring WebFlux". Disponível em `https://www.baeldung.com/spring-webflux-retry`