Integração de Sistemas

Reactor Core/Web Reactive

Bruno Sequeira - brunosequeira@student.dei.uc.pt
Rui Santos - rpsantos@student.dei.uc.pt

1 Introdução

Neste projeto fomos desafiados a desenvolver um servidor web e uma aplicação cliente que vai usar os serviços do servidor utilizando a estrutura *WebFlux*. O projeto inclui a criação de um servidor reativo que disponibilizará dados sobre proprietários (owners) e os seus animais (pets), bem como o desenvolvimento de um cliente capaz de utilizar esses dados da maneira mais eficiente. Temos como principais objetivos: O **Desenvolvimento Reativo** - A aplicação do servidor e do cliente devem ser construídas utilizando uma abordagem reativa, destacando-se o uso de WebFlux. **Operações CRUD no Servidor** - O servidor vai fornecer serviço básicos de CRUD para os dados dos owners e pets. **Eficiência e velocidade no cliente** - A realização de estratégias para lidar com grandes conjuntos de dados e otimizar as consultas para uma execução eficiente e rápida. **Tolerância a Falhas** - Vamos fazer uma query no lado do cliente que possa tolerar falhas da rede, implementando mecanismos de retry para se reconectar. Neste relatório vamos falar sobre a Base de Dados, serviços CRUD, servidor, Cliente - Reactive Code, Queries e as nossas abordagens, tolerancia a falhas, threads e sincronização.

2 Base de Dados

Para a implementação deste projeto, é necessário ter dados para que possamos testar no futuro as queries que iremos implementar, para isso criamos uma base de dados que contém duas tabelas owner e pet.

2.1 Tabela `owner`

```
CREATE TABLE IF NOT EXISTS owner (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    telephone_number TEXT NOT NULL
);
```

A tabela owner serve para guardar informações sobre os owners(proprietários). Esta possui três colunas:

- id: Uma chave primária, que será autoincrementada para cada novo owner.
- name: Uma coluna de texto que guarda o nome do owner.
- telephone number: Uma coluna de texto que guarda o número de telefone do owner.

2.2 Tabela `pet`

```
CREATE TABLE IF NOT EXISTS pet (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    species TEXT NOT NULL,
    birthdate DATE NOT NULL,
    weight INTEGER NOT NULL,
```

```
    owner_id INTEGER,
    FOREIGN KEY (owner_id) REFERENCES owner (id)
);
```

A tabela pet é criada para armazenar informações sobre os animais. Ela possui seis colunas:

- id: Uma chave primária, que será autoincrementada para cada novo pet.
- name: Uma coluna do tipo texto que guarda o nome do pet. species: Uma coluna do tipo texto que guarda a espécie do pet.
- birthdate: Uma coluna do tipo data que guarda a data de nascimento do pet.
- weight: Uma coluna que guarda o peso do pet do tipo inteiro.
- owner _id: Uma coluna de número inteiro que é uma chave estrangeira referenciando a coluna id da tabela owner.

A cláusula FOREIGN KEY estabelece uma relação entre a tabela pet e a tabela owner, garantindo que o owner_id na tabela pet corresponda a um id válido na tabela owner.

3 Operações CRUD

As operações CRUD desempenham um papel fundamental ao fornecer uma estrutura essencial para manipulação de dados, sendo que o acrónimo CRUD representa as quatro operações básicas que podem ser realizadas em qualquer sistema que lide com dados persistentes. As operações CRUD estão relacionadas com os controladores, sendo que implementam as operações CRUD relacionadas aos owners/pets, fornecendo endpoints RESTful para interação com o serviço OwnerService/PetService. Para isso realizamos estas operações, no owner:

- **Criação de Owner POST:** É usado o método *createOwner(Owner)*, este tal como o nome indica cria um owner.
- **Leitura de Owner por ID ou Leitura de todos os Owners GET:** É usado o método *getAllOwners()*,*getOwnerById(Integer Id)*, que retornam todos os owners e um owner em especifico respetivamente.
- **Atualização de Owner PUT:** É usado o método *updateOwner(Integer,id Owner)*, este tal como o nome indica atualiza a informação de um certo owner, consoante o id.
- **Eliminação de Owner DELETE:** É usado o método *deleteOwner(Integer Id)*, este tal como o nome indica elimina um owner com determinado ID, mas só irá realizar este pedido apenas se o Owner não estiver associado a um pet.

E no pet as operações são as seguintes:

- **Criação de Pet POST:** É usado o método *createPet(Pet)*, este tal como o nome indica cria um pet, sendo que tem de associar a um owner existente na base de dados.
- **Leitura de Pet por ID ou Leitura de todos os pets GET:** É usado o método *getAllPets()*, *getPetById(Integer Id)*, que retornam todos os pets e um pet em especifico respetivamente.
- **Atualização de Pet PUT:** É usado o método *updatePet(Integer,id Pet)*, este tal como o nome indica atualiza a informação de um certo pet, consoante o id.
- **Eliminação de Pet DELETE:** É usado o método *deletePet(Integer Id)*, este elimina um pet com determinado ID.

Essas operações formam a base para construção de funcionalidades mais complexas, ajudando-nos a obter informações de forma mais simples.

4 Servidor

O servidor é um servidor Spring Webflux que serve para expor serviços web. Tem uma estrutura tipica de uma aplicação WebFlux, com uma classe *IsprojectApplication* que é que tem a classe *main*, e depois temos várias pastas como a controller onde tem os controladores tanto para o Pet como para o Owner, responsáveis por tratar dos requests do cliente, temos a pasta entity com os ficheiros que representam as tabelas da base de dados, a pasta repository que tem os ficheiros responsáveis pelo acesso aos dados, e a pasta services, que contém as classes que tratam da lógica do sistema ligando os controladores e os repositórios. O servidor conta também com logging, que foi usado como forma de monitorização e ajuda ao debug do sistema

5 Cliente - Reactive Code

O cliente é uma aplicação independente do servidor que deve recolher informação do servidor de acordo com as queries fornecidas. As respostas a essas queries são guardadas em ficheiros de acordo com o nome da query (p.e query 3 é guardada em Results/resp3.txt). Sendo esta aplicação, como já mencionado, independente do servidor, temos classes a representar as entidades tanto do lado do servidor como do lado do cliente. Para além disso temos em ambos uma entidade PetStatistics que serve para nos auxiliar na resposta a algumas questões.

5.1 Queries

Para executar as queries, vamos usar um webclient que vai tratar dos pedidos de informação ao servidor.

Query 1: Names and telephones of all Owners: Foi criada a função *getOwnersNameAndPhone()* onde é pedido ao servidor toda a informação sobre os owners sobre a qual fazemos um map de forma a extrair a informação pedida, isto é, o nome e o número de telefone de cada owner, conectando-os numa única string.

Query 2: Total number of Pets: Aqui foi criada a função *getTotalNumberOfPets()* onde é feito um pedido de toda a informação dos pets ao servidor e usamos o método *.count()* para obter o número total de pets.

Query 3: Total number of dogs: Da mesma forma que na query anterior pedimos ao servidor toda a informação dos pets mas antes de aplicar o método *.count()* usamos o método *.filter()* para filtrarmos a informação de forma a obter o número total de animais cuja espécie é "Dog".

Query 4: Sort the list of animals weighting more than 10 Kg in ascending order of animal weight: Usamos a função *getAnimalsWithWeightGreaterThan(Integer weight)* onde mais uma vez vamos pedir a informação de todos os pets ao servidor e aplicamos um filtro para obter apenas os pets com um peso acima de um dado valor, neste caso 10kg. Depois aplicamos o método *.sort()* para ordenar os animais por ordem crescente do seu peso.

Query 5: Average and standard deviations of animal weights: Nesta query, usamos a função *getAverageStandardDeviationsOfAnimalWeights()* sendo que é feito o pedidio de toda a informação acerca dos pets, com isso usamos o collect que acumulda toda a informação, usamos uma classe *PetStatistics* esta que mantém as informações como o total, o contador e a soma dos quadrados dos pesos. Para cada Pet, os pesos são somados no total, o contador é incrementado e a soma dos quadrados dos desvios em relação à média é calculada. O resultado dessa operação é mapeado para o resultado final. O *result()* é um método da classe 'PetStatistics' que calcula a média e o desvio padrão com base nos valores acumulados.

Query 6: The name of the eldest Pet: Aqui voltamos a fazer um pedido de toda a informação dos pets ao servidor, onde vamos ordenar por ordem decrescente de data de nascimento, e sendo o último o mais velho, usamos o método *.last()* para o obter.

Query 7: Average number of Pets per Owner, considering only owners with more than one animal: Para este pedido, foi usada a função *calculateAverageOfPetsWithOwnersHaving-MoreThanOnePet()* onde fazemos um pedido de todos os pets agrupando-os pelos seus owners(a partir da foreign key), após isso, é feito uma contagem de pets em cada grupo e o resultado é filtrado para incluir apenas aqueles que têm mais que um pet. (O resultado é o contador de Pets para cada owner que apresente mais que um pet).
Juntando isso, esse resultado é reduzido usando a operação reduce, sendo que cada valor é acumulado numa instância 'PetStatistics', tal como foi usada na Query 5, que mantêm o total de pets e o total de owners. Com esse resultado, mapeamos o resultado calculando a média de pets, usando uma função que pertence à instância *average()*, retornado assim a média.

Query 8: Name of Owner and number of respective Pets, sorted by this number in descending order: Criamos uma função de nome *NameOfOwnerandNumberOfRespectivePets()*, onde fazemos o pedido ao servidor de todos os owners e pets. Após o pedido, mapeamos os owners para um fluxo de tuplos (owner, pet id) que associa os ID's dos pets ao owner. Isso é feito filtrando os pets pelo ID do seu proprietário, ou seja, se o ownerid do pet for o mesmo que o id do owner, _esse id é guardado, sendo que é no final ordenado de forma descendente a partir da operação *.sort()*. Finalmente o resultado do fluxo de tuplos, é mapeado para um fluxo de string onde cada string representa as informações do owner e os ID's de seus Pets Associados.

Query 9: The same as before but now with the names of all pets instead of simply the number: Para este pedido foi criada a função *NameOfOwnerandNameOfRespectivePets()* sendo que obtemos os todos os owners e pets. Em seguida, para cada owner é criado um fluxo contendo os nomes dos pets associados a esse owner, usando o mesmo método da query 8, filtrando a partir da foreign key do owner. Com isso é mapeado apenas para obter os nomes dos pets, e em seguida criamos um tuplo que contém o nome do owner e o do pet. Além disso, os resultados são ordenados por ordem decrescente com base nos nomes dos Pets. No fim, o fluxo de tuplos é mapeado para um fluxo de strings onde cada string tem as informações do owner e o nome do seu pet associado.

5.2 Tolerância a falhas de rede

Foi também pedido para que uma query conseguisse tolerar falhas de rede, repetindo o pedido várias vezes até desistir. Para demonstrar isto, criamos a função *getOwnersNameAndPhoneRetry()*. Aqui fazemos exatamente a mesma coisa que na query 1, adicionando apenas o método *.retry()*. Do lado do servidor, no controlador do owner, temos a função *getAllOwnersRetry()* onde com o uso de um contador, retornamos um erro de runtime enquanto não for o ultimo retry, e quando for o último, retornamos a informação pedida, que é a mesma da query 1.

6 Tratamento de Erros

No contexto do desenvolvimento, é crucial implementar estratégias eficazes para lidar com situações de erro. Para isso, imlementamos tanto no lado do servidor como no lado do cliente, métodos que garantam uma gestão eficiente de erros, proporcionando feedback claro e preciso sobre o estados das operações.

6.1 Lado do Servidor

Para o tratamento de erros do lado do servidor, usamos os métodos *.onErrorResume()* e *.doOn-Terminate()*. Foi criada uma variável atomica chamada de 'Error' inicializada com valor 'false', e será usada para controlar se ocorreu algum erro durante a operação. O primeiro método é utilizado para lidar com qualquer erro que ocorra durante o *.save()*, se ocorrer um erro, o 'Error' é marcado como true e um 'Mono.empty()' é retornado para indicar que o fluxo deve ser interrompido. O segundo método é usado independentemente de ocorrer um erro ou não. O valor do Error indicará se ocorreu um erro ou não, sendo que se for true/false, é registado um log informativo ou de erro indicando uma das seguintes opções (owner pode/não pode ser criado, owner pode/não pode ser eliminado, owner pode/não pode ser atualizado). Esses métodos são usados também para criação, atualização e eliminação de pets.

6.2 Lado do Cliente

No que toca à gestão de erros do lado do cliente, em cada query usamos os métodos *.onStatus()* e *.doOnError()*. O primeiro método retorna RuntimeException com uma mensagem de "Server Error" cada vez que há um erro nos pedidos Http. O segundo método serve para intercetar erros que ocorram durante o processamento do lado do cliente e se algo for detetado, imprime uma mensagem de erro indicando o tipo de erro.

7 Threads e Sincronização

Em termos de uso de threads, utilizamos a classe *CountDownLatch* da package *java.util.concurrent*. Esta classe serve para sincronizar threads e permite que uma ou mais threads fiquem à espera que todas acabem o trabalho. Inicializamos o CountDownLatch com o número de queries que vamos correr, ou seja, as nove queries pedidas e no metodo *.subscribe()* de cada uma chamamos o latch.countDown(), que é o contador do CountDownLatch. Este serve para no final, onde temos o *latch.await()*, que o programa espera que o CountDownLatch chegue a zero, que significa que todas as threads terminaram o seu trabalho. Depois, para criar as threads em si usamos o Scheduler da package *reactor.core.scheduler* para criar uma pool de 9 threads para correr em parelelo, e associamos cada query a uma thread através do *.subscribeOn()*. No final do programa chamamos o *dispose()* para libertar os recursos do scheduler.

8 Conclusão

Neste projeto, construimos uma aplicação web reativa utilizando a estrutura WebFlux. Para isso criamos um servidor que disponibiliza dados sobre owners e pets, assim como o desenvolvimento de um cliente capaz de pedir e tratar esses dados de maneira eficiente.

Ao longo deste relatório, discutimos a implementação de operações CRUD no servidor, que fornecem funcionalidades básicas de criação, leitura, atualização e exclusão de dados. Exploramos a base de dados, com tabelas para owners e pets, estabelecendo uma relação entre eles.

No lado do cliente, abordamos a implementação de queries reativas para pedir informações específicas ao servidor. Cada query foi projetada para atender a requisitos específicos do enunciado e para lidar com falhas de rede.

Além disso, destacamos estratégias de tratamento de erros, tanto no servidor quanto no cliente, para fornecer feedback claro e preciso em situações adversas. A tolerância a falhas de rede foi abordada com a implementação de uma query capaz de reagir a falhas, tentando reconectar-se várias vezes antes de desistir.

No aspecto de concorrência e sincronização, utilizamos threads e a classe CountDownLatch para gerir a execução paralela das queries. Isso resultou num melhor desempenho, especialmente ao lidar com queries mais demoradas.

Em conclusão, este projeto proporcionou uma experiência prática valiosa no desenvolvimento de aplicações reativas, destacando o tratamento de erros e considerações de concorrência para criar sistemas robustos e responsivos. A aplicação de conceitos como WebFlux e estratégias reativas contribuiu para a construção de uma solução moderna e eficaz.